BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

REMOTE API WEB REFERENCE FOR JAVA ENTERPRISE APPLICATIONS

TESTOVÁNÍ VZDÁLENÝCH APLIKAČNÍCH ROZHRANÍ JAVA ENTERPRISE APLIKACÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ONDŘEJ KRPEC**
AUTOR PRÁCE

SUPERVISOR **Ing. RADEK KOČÍ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

The Master's thesis focuses on testing REST API interfaces of web applications. The result of the thesis is the Restty application, which allows to test API endpoints of said interfaces, as well as to create extensive test cases from the interface's endpoints. The theoretical part of the thesis explains the principles of web services, remote interfaces, technologies that are used for development of the application, and the Swagger framework upon which the application is built. Subsequently, the design drafts of the application are presented in detail. The following section covers Restty's implementation and demonstrates how the Restty can be used for testing the interface of a Red Hat JBoss BPM Suite application. In conclusion, the benefits of the Restty application are evaluated and its possible extensions are proposed.

Abstrakt

Tato diplomová práce popisuje testování REST API rozhraní aplikací. Výsledkem práce je aplikace Restty, založená na použití nástroje Swagger, která umožňuje testovat jednotlivé části API aplikací, i vytvářet a spouštět komplexní testovací scénáře nad daným rozhraním. Teoretická část práce vysvětluje principy webových služeb, vzdálených rozhraní a představuje nástroj Swagger i technologie použité k implementaci. V následující kapitole jsou v práci prezentovány designové návrhy aplikace, na které plynule navazuje kapitola o implementaci a testování, pro které je zvoleno rozhraní nástroje Red Hat JBoss BPM Suite. V závěru práce jsou vyhodnoceny přínosy aplikace Restty a navrženy případné budoucí rozšíření.

Keywords

REST, Continuous testing, Automated testing, Swagger, Angular, Java, TypeScript, PostgreSQL, Spring Framework, Hibernate, PatternFly, Red Hat, Web Services, Test cases, Test automation

Klíčová slova

Webové služby, REST, Automatické testování, Průběžné testování, Swagger, Angular, Java, TypeScript, PostgreSQL, Spring Framework, Hibernate, Patternfly, Red Hat, Testovací scénáře

Reference

KRPEC, Ondřej. *Remote API Web Reference for Java Enterprise Applications.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Kočí, Ph.D.

Rozšířený abstrakt

Tato diplomová práce popisuje testování REST API rozhraní aplikací. Výsledkem práce je aplikace Restty, která je založená na použití nástroje Swagger, jenž umožňuje získat dokumentaci REST API rozhraní aplikací z jejich zdrojového kódu a zpřístupnit toto rozhraní v podobě JSON souboru. Cílem Restty je umožnit jejím uživatelům testovat jednotlivé části API aplikací a navíc i vytvářet a spouštět komplexní testovací scénáře nad daným rozhraním.

Teoretická část práce nejrpve vysvětluje principy webových služeb, vzdálených rozhraní, REST principy a především způsoby komunikace a přístupu ke zdrojům v rámci jednotlivých webových služeb pomocí metod, poskytovaných HTTP protokolem. Následně jsou v práci představeny některé, již existující aplikace, které jsou v současnosti používany pro testování REST rozhraní aplikací, jejich nedostatky a nevýhody.

V následující kapitole jsou představeny technologie použité pro implementaci aplikace. Uvedené technologie jsou rozdělené do tří částí. V první části jsou rozebrány technologie pro implementaci serverové části aplikace, konkrétně jazyk Java a standardní podporované knihovny pro tvorbu webových aplikací v tomto jazyce – konkrétně knihovna Spring, umožňující tvorbu samotné webové aplikace, a Hibernate, knihovna poskytující "objektově-relační" mapování. Ve druhé části jsou vysvětleny principy technologií, které jsou použity pro vývoj klientské části aplikace. Konkrétně se jedná o knihovnu Angular, která umožňuje vytvářet klientskou část webových aplikací pomocí jazyků HTML, CSS a TypeScript. V následujících odstavcích jsou vysvětleny principy již zmíněného jazyka TypeScript, jenž vznikl jako rozšíření jazyka JavaScript, a jeho role v použité knihovně Angular. V poslední řadě, se kapitola věnuje představení knihovny PatternFly, a jejím principům, která byla použita ke stylování aplikace. Třetí část kapitoly se věnuje výhradně popisu nástroje Swagger a jeho použití v aplikaci.

Následující kapitola obsahuje kompletní návrh aplikace Restty a popisuje s jakými cíly byla aplikace navržena. V prvních odstavcích jsou nejprve detailně probrány požadavky, které byly na výslednou aplikaci kladeny. Na jejich základech byl následně vytvořen databázový model celé aplikace, který je v kapitole detailně rozebrán a ilustrován ve formě ER diagramu. Na databázový model poté postupně navazují odstavce, jenž popisují designové návrhy aplikace, které byly použity jako předloha pro samotný vývoj klientské části aplikace.

Na návrhy aplikace plynule navazuje kapitola o implementaci a testování, ve které je nejprve probráno rozdělení aplikace na klientskou a serverovou část, konkrétně použití návrhového vzoru "Separation of Concerns", a následně struktura obou částí. Dále jsou v kapitola detailněji rozebrány důležité části implementace, konkrétně je v textu vysvětleno vytváření nových projektů v Restty a zpracování vstupního JSON souboru s API dokumentací, a také samotná implementace testování API a volání jejich HTTP dotazů. V poslední řadě je v diplomové práci probráno testování aplikace Restty, k čemuž bylo využito rozhraní aplikace Red Hat JBoss BPM Suite.

V závěru je celá práce shrnuta, jsou vyhodnoceny přínosy celé aplikace oproti stávajícím řešením a je navrženo několik dodatečných rozšíření, které umožní aplikaci ještě více šetřit čas vývojářům, kteří by jinak museli tento čas trávit manuálním psaním testů.

Remote API Web Reference for Java Enterprise Applications

Declaration

Hereby I declare that this master thesis was prepared as an original author's work under the supervision of Ing. Radek Kočí, Ph.D. The supplementary information was provided by Mgr. Ivo Bek. All the relevant information sources, which were used during preparation of the thesis are properly cired and included in the list of references.

.....
Ondřej Krpec
May 21, 2018

Acknowledgements

I would like to thank Mgr. Ivo Bek for his technical guidance of the master thesis. At the same time, I would like to thank Ing. Radek Kočí, Ph.D., for his pedagogical leadership.

Contents

Chapter 1

Introduction

In the software industry, the accessible and testable code is crucial for modern businesses, and the best way for developers to access or test it is through APIs[1]. APIs are supposed to connect engineers, let companies add value to their products and create an ecosystem of shared knowledge that allows other developers to use the functions provided by the interfaces. To fulfill these tasks, the interfaces have to be clear, accessible and, most importantly, human and machine readable. However, despite their importance, there hasn't been an industry standard for documentating nor testing them.

The thesis aims to solve the problem of testing the applications interfaces. The goal is to develop an application having an innovative user interface with regard to clarity and simple use, aimed at developers, even in the case of large interfaces. The application will be based on the Swagger framework, which provides a way to automate API reference generation. The resulting application, called Restty, will provide not only way to test single API endpoints, but mainly the functionality to create extensive test cases from the listed web services.

The thesis is organized as follows. Chapter 2 gives definitions needed to follow the thesis and explains web services and RESTful APIs in detail. Chapter 3 focuses on technologies that were used for the development of the Restty application. Chapter 4 describes the application's designs and mockups, which were used to reveal any clashing visual elements before writing the code. The penultimate Chapter covers the development of Restty using the technologies listed in previous sections and testing the application using the API from the Red Hat JBoss BPM Suite application. The last Chapter contains an overall summary of the developed solution and final thoughts on the work done within the thesis.

[1] API is an abbreviation for an Application Programming Interface which is a set of protocols and tools for building application software.

Chapter 2

Preliminaries and Definitions

This chapter will gradually introduce terms necessary to follow the thesis. In the first section basic terminology is introduced and established notion of remote interfaces and web services. In the next section an explanation of what APIs are is provided and the last section covers the importance of their testing.

2.1 Understanding the Web Services

A web service is a software system designed to support interoperable machine to machine interactions over a network. It is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over the networks like the Internet in a manner similar to interprocess communication on a single computer.

In the past, web services used mostly SOAP[1] over HTTP protocol [4], allowing less costly interactions over the Internet. However, in 2004 the W3C extended the definition of web services about "REST-compliant" web services [14], in which the primary purpose of the web service is to manipulate XML or JSON representations of web resources using a uniform set of stateless operations.

2.1.1 Introduction to RESTful Web Services

The REST, abbreviation of Representationl State Transfer, is an architectural style for networked hypermedia applications, primarily used to build web services. The term was first defined in the year 2000 by R. Fielding in his doctoral dissertation [3]. In the dissertation, Fielding explained that the REST principles were known as the "HTTP object model" beginning in 1994, and were used in designing the HTTP 1.1 and Uniform Resource Identifiers [1] standards.

The REST architectural style constrains an architecture to a "client-server" architecture and is designed to use a stateless communication protocol, typically HTTP. A client and a server exchange representations of resources by using standardized interface and a protocol. When the client accesses the resource using unique URI, a representation of the resource is returned. With each new resource representation, the client is said to transfer

[1] Simple Object Access Protocol is a protocol specification for exchanging structured information in the implementation of web services in the computer networks.

state. The resources are typically represented by text, JSON or XML, with JSON being currently the most popular format being used.

2.1.2 Messaging

As mentioned in the previous section, the RESTful web services can use any stateless communication protocol as a medium of communication between client and a server. However, the HTTP protocol is the most popular. The communication works as follows: the client sends a message in form of HTTP Request and the server responds in the form of HTTP Response. This technique is termed as *Messaging*. Apart from the data, the messages also contain some metadata about the message itself. As can be seen in the figure 2.1, a request message consists of five major parts.

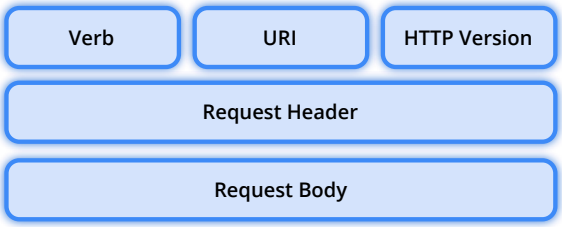

Figure 2.1: The format of a HTTP Request.

1. Verb indicates the HTTP method like GET, PUT, POST, etc.
2. URI is the Uniform Resource Identifier used to identify the resource on the server.
3. HTTP version is the version of HTTP.
4. Request header contains metadata as a collection of "key-value" pairs of headers and their values. For instance, a client (or browser) type, format supported by the client, format of the message body, cache settings for the response, and more.
5. Request body is the message content or resource representation.

In the listing 2.1 can be seen an example of a request that was created by the browser when it tried to access the website of Faculty of Information Technology.

```
1 GET / HTTP/1.1
2 Host: www.fit.vutbr.cz
3 User-Agent: Mozzila/5.0 (Windows NT 6.3; Win64; x64) ...
4 Accept: text/html,application/xhtml+xml,application/xml; ...
5 Accept-Encoding: gzip, deflate
6 Accept-Language: cs-CZ,cs;
```

Listing 2.1: An example of a simplified GET request made by the browser.

As can be seen, the HTTP method is followed by the URI and the HTTP version. The request also contains some headers. For instance the "*User-Agent*" header contains information about the type of a client which made the request. The *Accept* headers tells the server about various representation formats, the encoding, and a language the client supports. The server, if it supports more than one representation format, can decide the format for the response at runtime depending on the value of the *Accept* header.

When the server receives the request it responds with a HTTP response which consists of four major parts, as can be seen in the figure 2.2.

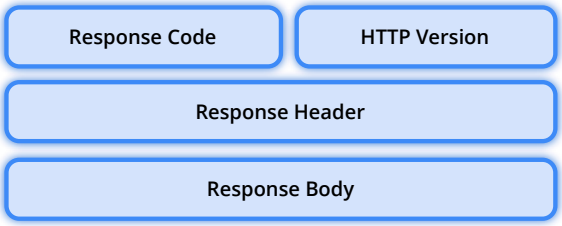

Figure 2.2: The format of HTTP response.

1. Response code contains the server status for the requested resource. The response is a "3-digit" status code, for instance, 404 means resource not found and 200 means response is ok.
2. HTTP version is the version of HTTP.
3. Response header contains metadata and settings of the response message as "key-value" pairs. For example, content type, content language, response date, etc.
4. Response body contains message content or resource representation if the request was successful.

In the listing 2.2 can be seen an example of a response to a request from the listing 2.1. The response contains the version of HTTP, response code and several response headers followed by the response body which in this case is a HTML page.

```
HTTP/1.1 200 OK
Date: Sat, 09 Dec 2017 08:36:01 GMT
Server: Apache
Content-Location: index.php.cz
Pragma: no-cache
Keep-Alive: timeout=60, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-2
Content-Language: cs
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
...
```

Listing 2.2: An example of a simplified response to GET request.

2.1.3 Addressing the Resources

If the client wants to get a resource from the server, it needs to know, how to address it. *Addressing* refers to locating a resource or multiple resources lying on the server. It is analogous to locating a postal address of a person. A RESTful service uses directory hierarchy like human readable URIs to address its resources. Each resource is identified by its URI which is of the following format:

$$< protocol >: // < serviceName > / < resourceType > / < resourceId > \quad (2.1)$$

However, it needs to be noted that the URI should not say anything about the operation or action, because for identifying the operation to be performed on the resource, the HTTP verbs are used. This enables the client to call the same URI with different verbs to perform different operations. The verbs correspond to read, create, update, and delete[2] operations. Nevertheless, while designing URIs there are other practices that should be considered as well.

1. Plural nouns should be used to define resources.
2. Spaces should be avoided. Typically in URIs it's recommended to use underscore or hyphen when using long resource name.
3. Lowercase letters are recommended. Although URIs are "case-insensitive", it's a good practice to keep them in lower case letters only.
4. Avoid verbs for the resource name until the resource is actually an operation or a process.

2.1.4 HTTP Verbs

As stated in section 2.1.3 the most used HTTP verbs correspond to CRUD operations. For instance, read operation corresponds to the GET verb which is used to retrieve a representation of a resource. If the request is successful, it returns representation of the resource in a format that is accepted by the client and a response code of 200. Note that the requests that utilize the verb should be used only to read data, not change it. When used this way they are considered safe. That is, they can be called without risk of data modification or corruption. Additionally, the requests are idempotent, which means that making multiple identical requests ends up having the same result as a single request.

For creating resources, the POST is most often utilized. On successful creation, returning a *Location* header, with a link to the created resource, and the 201 response code. The method is neither safe nor idempotent. Therefore it is recommended for "non-idempotent" resource requests. Making two identical POST requests will most likely result in two resources containing same information.

For update capabilities, the PUT verb is ofter most utilized, "PUT-ing" to a known resource with the request body containing the newly updated representation of the original resource. On successful update, the server should return the 200 or 204 status code if not returning any content in the body. It follows from the above that a body in the response is optional – providing one is not necessary and only leads to more bandwidth consumption. The PUT is not a safe operation, in that it modifies state on the server, but is idempotent. In other words, if the resource is updated using the PUT method and then the same call is made again, the resource is still there and has the same state as it did with the first call.

The DELETE verb is pretty straightforward, it is used to delete a resource. On successful deletion, the server should return response code of 204 with no response body. "HTTP-spec-wise", the operations are idempotent. If the resource is deleted, it is gone.

[2]In computer programming, create, read, update, and delete (as an acronym CRUD) are four basic functions of persistent storage.

Repeatedly calling the method on that resource ends up the same – the resource is gone. However, there is a caveat about the method's idempotence. Calling it on the resource a second time will result in 404 response code since the resource was already removed and therefore is no longer findable. It makes the operation in fact no longer idempotent, however, the "end-state" of the resource is the same.

2.1.5 Representation of the Resources

It is clear that the focus of RESTful services is on resources and on providing access to them. A resource can easily be thought of as an object as in OOP[3]. The resources can be text files, HTML pages, images or videos and can consist of other resources. The server simply provides access to the resources and client accesses and modifies them. It is important to point out that the architecture does not put a restriction on the format of a resource representation. However, as mentioned before, the most popular representation formats are XML and JSON.

```xml
<user>
  <id>1</id>
  <firstName>John</firstName>
  <lastName>Doe</lastName>
  <age>42</age>
</user>
```

Listing 2.3: An example of a XML representation of a *user* resource.

Once a resource is identified then its representation is to be decided using a standard format so that the server can send the resource in the above said format and the client can understand said format.

```json
{
  "id": 1,
  "firstName": "John",
  "lastName": "Doe",
  "age": 42
}
```

Listing 2.4: An example of a JSON representation of a *user* resource.

Despite the fact that there are no restrictions on the format of a resource representation, following some important points should be considered. For instance, both the server and the client should be able to understand said format. Moreover the format should be able to represent the resource completely.

2.2 Introduction to Application Programming Interfaces

In computer programming an application programming interface is the defined interface through which interactions happen between an enterprise and users of its assets. It can become the primary entry point for enterprise service, for its own website and applications, as well as for a partner and customer integrations. It is defined through a contract so that any application can use it with relative ease.

[3]Object oriented programming (OOP) is a programming paradigm based on the concept of *objects*, which may contain data, in the form of fields; and code, in the form of procedures, often known as methods.

The API [8] approach creates a loosely coupled architecture that allows a component service to have a wide range of future uses, and is technology agnostic. The architecture resolves around providing programmable interfaces to a set of services to different applications serving different kinds of customers. It assumes that these user groups might change or evolve over time in the way they utilize the provided services. The strategy of providing APIs leads to the following benefits:

1. With APIs, computers rather than people can manage the work. Through them, companies can update work flows to make them quicker and more productive.

2. They allow content to be embedded from any site or application more easily. This guerantees more fluid information delivery and an integrated user experience.

3. Using APIs, any user or company can customize the content and services that they use the most.

4. They represent a cheaper way of building applications by increasing the reuse of services. Providing a usage or "analytics-based" evolutionary development platform decreases cost of development and change to services.

5. The company that releases the API allows its customers to access their conferencing services in new, more efficient ways, increasing brand recognition and customer loyalty.

2.2.1 When is API RESTful?

The previous sections explained the principles of RESTful web services and introduced the term Application Programming Interface. However, it is necessary to point out that not all web APIs are considered RESTful. An API is RESTful only when it is acting under the REST constraints at all times. These constraints restrict the ways that the server may process and respond to client requests so that, by operating within these constraints, the service gains desirable "non-functional" properties, such as performance, scalability, portability and reliability. These formal constraints are as follows:

1. **"Client-Server"** – the constraint is based on the separation of concerns principle. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms.

2. **Stateless** – communication between client and server have to be stateless. It means that each request from client to server must contain all the necessary information to complete the transaction. The main advantage of this approach is that the system is able to scale better because the server does not have to store client state between requests.

3. **Cacheable** – the constraint ensures that the clients can cache response to improve performance. "Well-managed" caching partially or completely eliminates some "client-server" interactions, further improving scalability and performance.

4. **Uniform Interface** – in order to have efficient caching in a network, components have to be able to communicate via a uniform interface. The definition of uniform interface consists of four other constraints, however most of them can be found implemented in the HTTP protocol.

5. **Layered System** – in a layered system, intermediaries, such as proxies can be placed between client and server utilising the web's uniform interface. The main advantage is that intermediaries can then intercept "client-server" traffic for a specific purposes; for example caching.

6. **Code On Demand** – it is an optional constraint and it allows clients to download programs for "client-side" execution. The best examples for this are compiled components such as Java applets or "client-side" scripts such as JavaScript.

2.3 The Importance of API Testing

Software testing is an important phase of the software development life cycle in general, with API testing being one of its most challenging parts [10]. It is being increasingly recognised as being more suitable for test automation and continuous testing than other forms of testing. Many developer teams are starting to increase the level of API testing while decreasing their reliance on GUI testing because the tests at the API layer are less brittle and easier to maintain even thought they have to cover individual functionalities as well as series or chain of functionalities.

In general API testing is used to determine whether the APIs and the integrations they enable work in the most optimal manner e.g. whether they return the correct response, react properly to edge cases, delivery responses in an acceptable amount of time, and respond securely to potential security attacks. In particular, the testing concentrates on using software to make API calls in order to receive an output before observing and logging the system's response. This enables to test if the API returns a correct response or output under varying conditions. The output is typically a pass or fail status, date, information or a call to another API. However, it is important to point out that there could be no output at all or something completely unpredicted can occur.

Overall, it's very clear that the risk of putting a bad and especially insecure product on the market is greater than the cost of testing it which is why the API testing is crucial part of the application development process.

2.3.1 Beginning with cURL

As APIs are becoming an integral part of how software works it is unsurprising that many frameworks and applications for theirs testing were developed. Probably the oldest of them is cURL [13]. It is a command line tool for transferring data using various protocols. It consists of two products – libcurl and curl. Libcurl is a "client-side" transfer library with support for a wide range of protocols. It is portable, "thread-safe", feature rich, and well supported on virtually any platform. On the other hand, curl is a command line tool for getting or sending files using URL syntax. Since curl uses libcurl, it supports the same range of common Internet protocols that libcurl does. In general, it provides a generic, language agnostic way to demonstrate HTTP requests and responses.

In addition, as REST follows the same model as the web, it is possible to type an HTTP address to the curl and use it to make an HTTP request to a resource on a server. The server returns a response, which would typically be converted by the browser to a more visual display, as a raw code to show the developers what they are really retrieving. Obviously, the requests that can be made with the tool may test various functionalities such as sending

requests using various HTTP verbs, specifying query strings and parameters or even using authentication.

```
curl --request POST \
  --url https://localhost:8080/users \
  --header 'authorization: Bearer {{AcessToken}}' \
  -d '{
    "firstName": "John", \
    "lastName": "Doe", \
    "age": 42 \
  }'
```

Listing 2.5: An example of POST request that creates *user* resource on the server using API endpoint */users*.

However, it is a little cumbersome to work directly with curl, since even a simple curl request may look like in the listing 2.5. Therefore, other frameworks and applications, such as Postman, were developed.

2.3.2 Continuous Testing with Postman

Postman is a useful tool for testing the functionality of API endpoints. It has a nice UI, which makes it easy to add or remove parameters, define headers, authorization methods, and data without the hassle of writing code. It also allows developers to create various environments, variables, and to save requests, which curl is not designed to do.

Besides providing a friendly user interface for constructing HTTP requests, Postman also gives developers the ability to write tests against the responses of requests to see if the server is returning the correct results. Requests constructed in Postman can also be bundled into a collection and easily exported or shared, making Postman great for collaborating on and sharing API specifications with other developers. In addition, the collections can also be used with continuous integration systems so that the same collection used to test an API locally while developing can also be used to determine whether or not the codebase should be pushed live onto production.

However, there is a problem with using Postman for continuous testing, because as was stated in the section 2.1.4, not all HTTP verbs are idempotent. To clarify, what if a developer creates a test that removes a resource with specified identifier from a database? If the code covered by the test is bugfree then the resource is removed and the test successful. Nevertheless, running the test repeatedly will result in the test's failure as the resource no longer exist. The solution would be to create a test case that calls the endpoint which creates the resource, and then using the resource identifier from the response to remove it. Unfortunately, Postman or any other similar application does not allow to use request's response as an input of another request.

Chapter 3

Technologies and Frameworks

In this chapter the details of technologies and frameworks that were used for the development of the Restty aplication are discussed. In the first sections the Java language and the Spring and Hibernate frameworks that were used for building the backend of the application are introduced. Following the thesis the frontend web application framework Angular along with the superset of EcmaScript 6 – the TypeScript language is presented, as well as the web framework PatternFly and its key features. Finally, the last section covers the Swagger framework and its usage within the Restty application.

3.1 Introduction to Java

The Java language project was initiated in June 1991 by J. Gosling [7] for use in one of his "set-top" box projects. The language, initially called *Oak*, ended up later being renamed as Java when it was first publicly released by Sun Microsystems in 1995. The language promised "Write Once, Run Anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilations, as illustrated in the figure 3.1.

The WORA is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of compiling the code directly to architecture specific machine code. When compiled, the bytecode is executed by a Java Virtual Machine, which is a separate program that is optimized for the specific platform on which the Java code is run.

However, the Java's portability is not its only advantage over other languages that can be used to build backend of a web application. For instance, among the other benefits belongs the automatic storage management, strong typing, flexible namespace, or standards for connectivity to relational databases.

To clarify, automatic storage management means that the JVM automatically performs all memory allocation and deallocation while the program is running. The developers cannot explicitly allocate memory for new objects or free memory for objects that are no longer referenced. Instead, they depend on a JVM to perform these operations. The process of freeing memory is known as garbage collection.

In addition, Java's flexible namespace makes it perfect for writing complex applications. That is because Java defines classes and places them within a hierarchical structure that mirrors the domain namespace of the Internet, which avoids name collisions and allows Java applications to be distributed.

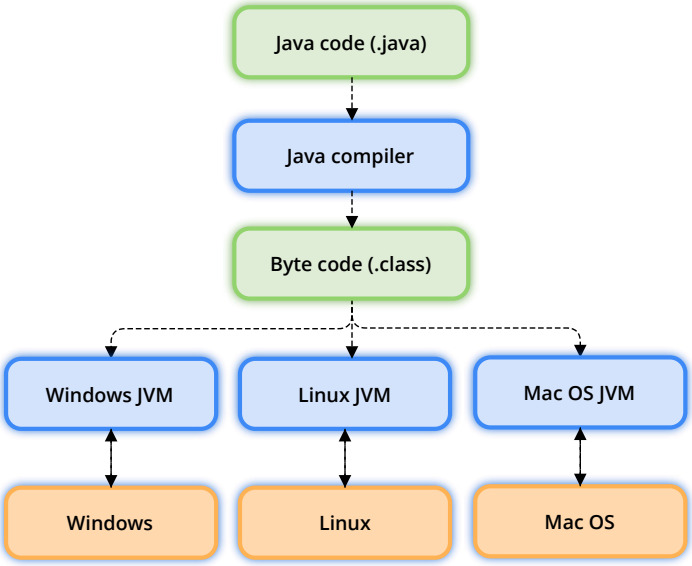

Figure 3.1: Illustration how Java ensures the "Write Once, Run Anywhere" property.

3.1.1 Basics of Java's syntax

The syntax of Java is largely influenced by C++. However, unlike C++, which combines the syntax for structured, generic, and "object-oriented" programming, Java was built almost exclusively as an "object-oriented" language. All code is written inside classes, and every data item is an object, with the exception of the primitive data types such as integers, characters and boolean values, which are not objects for performance reasons.

Following the above, a Java program can be defined as a collection of objects that communicate via invoking each other's methods. To clarify, object is an instance of a class, which is a template that describes the behavior and state that the object of its type supports. The behavior is described via methods, where the logics are written, data is manipulated and all the actions are executed.

Nevertheless, as was mentioned above, Java was influenced by C++ therefore it is not a surprise that the syntax is similar. However, there are some differences.

1. Java is case sensitive, which means identifier *Hello* and *hello* would have different meaning in Java.

2. For all class names, the first letter should be in upper case. If several words are used to form a name of the class, each inner word's first letter should be in upper case.

3. All method names should start with a lower case letter. If several words are used to form the name of the method, then each inner word's first letter should be in upper case.

4. The name of the class file should exactly match the class names. That is because if the file name and the class name do not match, the program will not compile.

Overall the Java's syntax and best practices are quite extensive, starting with basics like identifiers, modifiers, arrays or interfaces and ending with more advanced features like generics, annotations and lambda expressions that were added to the language specification

over the years. Unfortunately, these specifications are not the subject of the thesis to be explained in detail.

3.2 Building a RESTful Web Service

Even though Java is powerful language, building a backend of an enterprise application in plain Java is no easy task. Fortunately, there are many libraries and frameworks that provides a way of creating high performing, testable web applications with ease. One of the most popular is the Spring Framework [2], which is an open source Java platform, initially released in June 2003. Spring can be used for development of any Java application, but shines when used for building web applications on top of the Java EE platform. Spring targets to make J2EE development easier to use and promotes good programming practices by enabling a "POJO-based[1]" programming model.

3.2.1 The Advantages of Using Spring Framework

First and foremost, the technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control (IoC). The IoC is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of IoC.

When writing complicated Java application, the application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent. To clarify, consider an application which has a text editor component and the developer wants to provide a spell check to said component. The standard code would look similar to the code in the listing 3.1, creating a dependency between *TextEditor* and the *SpellChecker* classes.

```
1 public class TextEditor {
2
3   private SpellChecker spellChecker;
4
5   public TextEditor() {
6     this.spellChecker = new SpellChecker();
7   }
8
9 }
```

Listing 3.1: The standard way of providing spell checking to a component.

As can be seen in the listing 3.2, in an inversion of control scenario, the developer would create code differently, in this scenario the total control was removed from the *TextEditor* class and the dependency (i.e. *SpellChecker* class) is being injected into the *TextEditor* through a class constructor. Thus the flow of control has been "inverted" by Dependency Injection because the developer effectively delegated dependencies to an external system.

[1] In software engineering, a Plain Old Java Object (POJO) is an ordinary Java object, not bound by any special restriction and not requiring any class path.

```
public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

}
```

Listing 3.2: The Inversion of Control scenario of providing a *SpellChecker* to the *TextEditor* component.

The other key advantage of Spring is the Aspect Oriented Programming (AOP) framework. AOP entails breaking down program logic into distinct parts called concerns. The functions that span multiple points of an application are called "cross-cutting" concerns and are conceptually separate from the application's business logic. As mentioned earlier, the key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect, which is the combination of the pointcut and the advice [2]. As DI helps decouple the application objects from each other, AOP helps decouple "cross-cutting" concerns from the objects that they affect. To clarify, the AOP module provides interceptors to intercept an application meaning when a method is executed, the developers have the option to add an extra functionality before or after the method execution using the interceptors.

3.3 Persisting Data With Hibernate

Even though the Spring Framework covers a lot of functionality needed to build complex web application and adds significant enhancements to the data access layer of Java language, it is still best to use external ORM[3] framework. For the development of the Restty application I've chosen to use the Hibernate framework [9] on top of the H2 database.

Hibernate is a high performance ORM solution for Java, created by G. King in 2001. Hibernate maps Java classes to database tables, Java data types to SQL data types and relieves the developer from most of common data persistence related programming tasks. Hibernate consists of layered architecture which helps the developer to operate without having to know the underlying APIs, as illustrated in the figure 3.2. It makes use of the database and configuration data to provide persistence services (and persistent objects) to the application. The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. To provide such functionality, the classes should follow specific rules.

1. A class that will be persisted needs a default constructor.
2. All classes should contain an ID in order to allow easy identification of the objects within Hibernate and the database. The ID property is mapped to the primary key of a database table.
3. All attributes that will be persisted should be declared private and have appropriate getters and setters defined in the JavaBean style.

[2] Advice describes a class of functions which modify other functions when the latter are run.

[3] "Object-relational" mapping (ORM) is a programming technique for converting data between incompatible type systems using "object-oriented" programming languages.

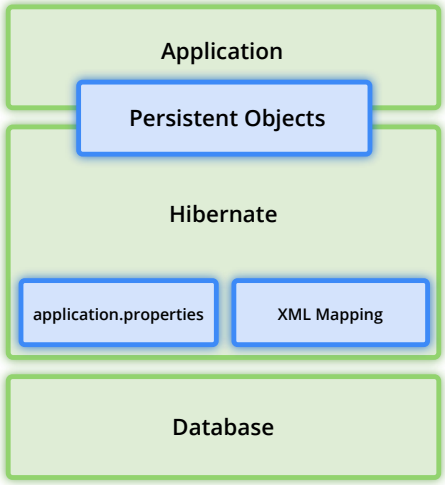

Figure 3.2: High level view of the Hibernate framework architecture.

As can be seen in the listing 3.3, a class is mapped to the database using Hibernate's annotations on the entity. Hibernate's annotations are a way of providing the metadata for the ORM mapping. All the metadata is clubbed into the POJO Java file along with the code, which helps the developer to understand the table structure and POJO simultaneously during the development.

```
@Entity
@Table("users")
@SequenceGenerator(name = "users_id_sequence", allocationSize = 1)
public class User {

  private Long id;
  private String name;

  @Id
  @Column(name = "id")
  @GeneratedValue(strategy = SEQUENCE, generator = "users_id_sequence")
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  @Column(name = "name")
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

}
```

Listing 3.3: POJO class that features the advantages of ORM.

When mapping the entity, Hibernate detects the annotations and accesses the properties through getter and setter methods by default. The primary key of the table is determined by the *@Id* annotation, which by default will automatically determine the most appropriate primary key generation strategy to be used. However, the default generation strategy can be overriden by applying the *@GeneratedValue* annotation, which takes two parameters – strategy and generator.

```
CREATE TABLE public.users
(
  id bigint NOT NULL,
  name character varying(255,
  CONTRAINT users_pky PRIMARY KEY (id)
)
```

Listing 3.4: The SQL code that is generated by Hibernate when mapping the entity from the listing 3.3.

The second advantage of using Hibernate is Hibernate Query Language (HQL) which is an "object-oriented" query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. The queries are translated by Hibernate into conventional SQL queries, which in turns perform an action on the database. Although it is possible to use SQL statements directly with Hibernate's Native SQL, it is not recomennded because of possible database portability hassles and Hibernate's SQL generation and caching strategies.

```
StringBuilder hql = new StringBuilder(" FROM ");
hql.append(User.class.getName());
hql.append(" WHERE name = :name ");

Query query = session.createQuery(hql.toString());
query.setString("name", name);
return query.list();
```

Listing 3.5: An example of HQL query that selects users with name "John Doe".

3.4 What is Angular?

Angular [5] is an open source TypeScript framework used to build web applications in HTML and TypeScript. It makes it easy to build an application as it combines declarative templates, dependency injection, "end-to-end" tooling, and integrated best practices to solve development challenges.

3.4.1 Beginning as AngularJS

Angular originally started as AngularJS, it was developed in 2009 by M. Hevery [6] as the software behind an online JSON storage service that would have been priced by the megabyte, for "easy-to-make" applications for the enterprise. However, the business idea was soon abandoned and AngularJS was released as an open source library in October 2010.

The framework was used to overcome obstacles encountered while working with Single Page applications[4]. However, because some of the core assumptions in AngularJS needed

[4] A single page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server.

to be changed, in September 2016 saw the light of the day a complete rewrite of AngularJS. Originally, the rewrite was called "Angular 2" by the team that built it, but this led to confusion among developers. To clarify, the team announced that separate terms should be used for each framework with "AngularJS" referring to the 1.X versions and "Angular" without the "JS" referring to version 2 and up.

As the new version of the framework was developed, some new concepts appeared. In addition to better "event-handling" capabilities, powerful templates, and better support for mobile devices, Angular introduced several new features.

1. The earlier version of Angular had a focus of controllers, but now has changed the focus to having components over controllers. Components help to build applications into many modules which helps in better maintaining the application over a period of time.

2. The newer version of Angular is based on TypeScript which is a superset of JavaScript, maintained by Microsoft. More information about TypeScript will be provided later in the section 3.5.

3. The newer version of Angular introduced services which are a set of code that can be shared by different components of an application. For instance, consider a data component that picks data from a database, it is possible to have it as a shared service that could be used across multiple components.

3.4.2 Angular's Core Concepts

As mentioned in previous section, Angular introduces the two core concepts – components and services, respectively the dependency injection. An Angular application will always have a root component that contains all other components. In other words, an application will always have a component tree, in which components are a logical piece of code that consists of following parts:

1. Templates that are used to render the view for the application. They contains the HTML that needs to be rendered as well as the bindings and directives.

2. Classes that are like a classes defined in any language, such as C, except they are defined in TypeScript. Classes contains properties, methods, and the code which is used to support the view.

3. Metadata that contains an extra data defined for the Angular class. They are defined using a decorator.

To clarify, consider an example from the listing 3.6 which contains all three parts. It defines a class called *HelloWorldComponent* which contains only one property – *title*. The component is then defined using the *@Component* decorator that contains HTML template which is the view that needs to be rendered in the application.

```
@Component ({
  selector: 'my-app-hello-world',
  template: `
    <div>
      <h1>{{title}}</h>
    </div>
  `,
})
export class HelloWorldComponent {
  title: string = 'Hello World!';
}
```

Listing 3.6: An Angular class with the *@Component* decorator and a HTML template.

The second cornerstone of an Angular application is dependency injection. The idea behind it is pretty simple, as illustrated in the figure 3.3. If a component that depends on a service is needed, the developers do not create the service by themselves. Instead they request one in the constructor and the framework will provide one. This approach leads to more decoupled code which enables testability and easier maintainence.

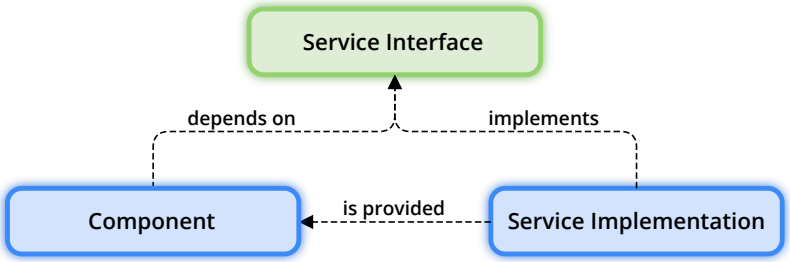

Figure 3.3: Angular's way of injecting and instantiating services in the components.

To clarify, consider an example from the listing 3.7. The listing contains a simplified service that has a method that returns an array of users. If a component is created and an argument of type *UserService* is passed to the component's constructor as a parameter, Angular automatically instantiates and injects the service into the component.

As can be seen, the Angular's dependency injection module is flexible, and easy to use because the objects can be injected only via constructors. In addition, the injectors form a hierarchy, and the injectable object does not have to be an "application-level" singleton as it might by default in Spring framework.

```
export class UserService {
  users: User[] = [];

  findUsers(): User[] {
    // Code used to retrieve the users
    return users;
  }
}


@Component {
  ...
}
export class UsersComponent {
  users: User[] = [];

  constructor(userService: UserService) {
    this.users = userService.findUsers();
  }
}
```

Listing 3.7: An example of dependency injection in Angular.

3.5 Introducing TypeScript

In september 1995 JavaScript was first introduced as a language for the client side. It was used to make webpages interactive and to provide online programs, including video games. However, as JavaScript code grows, it tends to get messier, making it difficult to maintain and reuse. Moreover, its failure to embrace the features of Object Orientation, strong type checking and "compile-error" checks prevent JavaScript from succeeding at the enterprise level as a "full-fledged" server side technology. TypeScript was presented to bridge this gap. Its main goals were to provide an optional type system and planned features from future JavaScript editions to current JavaScript engines.

3.5.1 Why Add Types to JavaScript

Types have proven ability to enhance code quality and understandability. Increasing the agility when doing refactoring and being one of the best forms of documentation a developer can have. However, types have a way of being unnecessarily ceremonious. Therefore TypeScript is very particular about keeping the barrier to entry as low as possible, only providing compile type safter for the JavaScript code. The great thing is that the types are completely optional.

TypeScript provides data types as a part of its optional type system, as illustrated in the figure 3.4. As a super type of all types, the *any* data type is used. It denotes a dynamic type and using it is equivalent to opting out of type checking for a variable. That suggests that the variable may be declared with no type which means that the type of the varible will be inferred by the TypeScript Language Service. All other "built-in" types and "user-defined" types inherit from the *any* type.

It is important to mention that the "built-in" types *undefined* and *null* may look similar but are not the same. A variable initialized with *undefined* means that the variable has no

value or object assigned to it, while *null* means that the variable has been set to an object whose value is undefined.

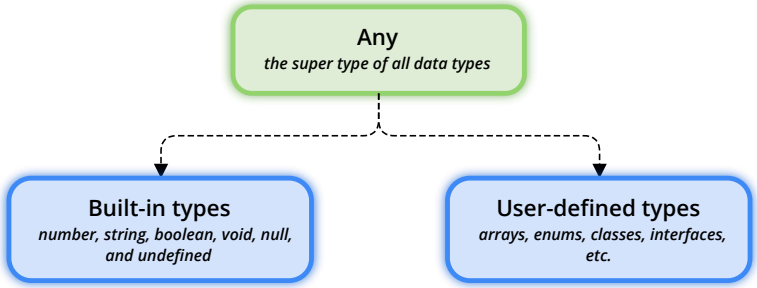

Figure 3.4: Classification of data types in TypeScript.

The main advantage is however that the JavaScript code files with the ".js" suffix can be renamed to a files with ".ts" suffix and TypeScript will give back a valid equivalent to the original JavaScript file. That is because TypeScript is intentionally and strictly a superset of JavaScript with optional type checking.

3.5.2 Future JavaScript

The second goal of TypeScript was to provide planned features from future JavaScript editions. Nowadays, it provides a number of features that were planned in EcmaScript 6[5] for current JavaScript engines. For instance, the language features modules and "class-based" orientation as well as features like generics and type annotations that are not part of the specification.

In conclusion, even thought we could use JavaScript with Angular, TypeScript feels like a superior choice, not only because it is strongly typed and supports object oriented programming, but because of the TypeScript's transpiler which provides the "error-checking" feature. Unlike JavaScript, the TypeScript is not an interpreted language and will compile the code and generate compile errors if it finds some sort of syntax errors. Thus helps to highlight the errors before the script is run hence, saving time trying to find the bugs in the code.

3.6 Styling with PatternFly

The success of an application depends on a "well-designed" user interface. The good or bad design can influence the perceived usability of an application, and if the application's design is not done well, the whole application can be badly perceived. The PatternFly framework was developed specifically to address this issue.

One of the main things that sets the framework apart from other libraries, such as Bootstrap, is the focus on design for IT enterprise applications. It recognizes the importance for a user to be able to migrate seamlessly from one product to another without having to relearn the UI. Behavioral consistency leads to better usability because users are familiar with the interactions. Visual consistency establishes a look and feel that users recognize and allows it unify disparate projects, make them look great and make them look like they belong in the same portfolio.

[5]The EcmaScript specification is a standardized specification of a scripting language.

PatternFly is an open source project that is based on Bootstrap [12], a "mobile-first" frontend framework for creating web sites and applications. It is developed using Less, a cascading style sheet "pre-processor" that extends the CSS language and adds features that allow variables, mixins, functions, and other techniques that allow developers create code that is maintainable, themeable and extendible. This allows the developers to add any required "app-specific" CSS directly into one CSS file, which is more performant, and make any necessary adjustments to PatternFly via variable overrides.

The framework consists of a series of Less stylesheets that implements various components of the toolkit. The stylesheets are generally compiled into a bundle and included in the applications, however individual components can be included or removed. Moreover, the framework provides a number of variables that control styling of various components. Each component consists of a HTML structure and CSS declarations, and in some cases accompanying JavaScript code.

3.6.1 Using the Components

The framework comes by default with various design templates for typography, tables, forms, buttons, and other interface components that can be used building the application – saving lots of time and efforts in the development process. The templates are made available as "well-factored" CSS classes that the developers can apply to the HTML to achieve different effects. By using semantic class name like *.alert* or *.alert-success*, the components are easily reusable and extensible. Although PatterFly uses descriptive class names that have a meaning, it is not specific about implementation details. Therefore all classes can be overriden with custom style and still, the meaning of the class will remain the same.

```
<div class="container">
  <div class="alert alert-success">
    <span class="pficon pficon-ok"></span>
    <strong>Hello World!</strong>
    <span>This is an example of an alert in PatternFly.</span>
  </div>
</div>
```

Listing 3.8: An example of styling the component using predefined class *.alert*.

As can be seen in the figure 3.5, the code snippet from the listing 3.8 generates a component that contains the "Hello World!" text. Using the semantic class names, the code is easily styled, allowing the developer to spend more time on application specific features and functions rather than application designs.

Figure 3.5: Illustration of a component styled using the semantic classes from PatternFly.

3.6.2 Working with the Grid

In the beginning of this section it was mentioned that the PatternFly, respectively its predecessor Bootstrap, was developed with a "mobile-first" design philosophy, which resulted in a framework that is responsive by design. The end result is that it easily and efficiently

scales with a single code base, from phones, through tablets, to desktops. The responsiveness is achieved using a fluid grid system that can be applied to appropriately scale up to 12 columns according to the size of the device or viewport. The grids provides structure to the layout, defining the horizontal and vertical guidelines for arranging content and enforcing margins.

As can be seen in the listing 3.9, to use the grid system, a few rules have to be followed. Grid column elements have to be placed inside row elements, which creates horizontal groups of columns. It is possible to have as many rows as needed, but it is necessary that the columns are immediate children of rows. In a full row, the column widths are any combination that adds up to 12, however it is not mandatory to use all available columns.

```
<div class="container">
  <div class="row">
    <div class="col-md-6">First column</div>
    <div class="col-md-6">Second column</div>
  </div>
  <div class="row">
    <div class="col-md-3">First column</div>
    <div class="col-md-3">Second column</div>
    <div class="col-md-3">Third column</div>
    <div class="col-md-3">Fourth column</div>
  </div>
</div>
```

Listing 3.9: An illustration of the grid system in PatternFly.

The rows have to be placed in a "fixed-width" layout wrapper that has a *.container* class attached and a width of 1170px or in "full-width" layout wrapper, which has *.container-fluid* class attached, and which enables the responsive behavior in that row. The grid system is based on four tiers of classes – *xs* for phones, *sm* for tables, *md* for desktops, and *lg* for larger desktops. These classes define the sizes at which the columns collapse or spread horizontally.

3.7 Introduction to Swagger Framework

Swagger [11] is an open source framework for designing and describing APIs. It was developed by Reverb Technologies in 2010 to solve the need for keeping the API design and documentation in sync. It provides a large ecosystem of tools that helps developers design, build, document, consume, and test RESTful web services. It defines a standard, language agnostic interface to APIs which allows both humans and machines to discover and understand the capabilities of the service. The standard is called the OpenAPI Specification which is a specification for "machine-readable" interface files. The files are essentially a resource listings of the API which adheres to the specification. The files are either of YAML or JSON format and contains a detailed description of the entire API. Nowadays there are two ways to create such file.

1. "Top-down" approach, or "design-first" which means using Swagger to design the API before writing any actual code.

2. "Bottom-up" approach, or "code-first" which means using Swagger to document the API of an existing code.

3.7.1 Using the Swagger

In the past, it was popular to use the "code-first" approach which is much easier because the developers can make adjustmens on the fly, and it fits nicely into an Agile delivery process. But because very often, the developers are not thinking about the design, it can make the API difficult to understand and document. To solve this, Swagger supports various annotations, as can be seen in the listing 3.10, that allow developers to specify the details of the documentation. The alternative way is to let Swagger figure out the documentation by itself based on the annotations from Spring framework[6]. Under the hood, it scans Spring controllers on "start-up" and registers a documentation controller that exposes the operations Spring controllers implement. The documentation follows the specification – any client that understands the specification can use the API. The important thing is that the documentation is based on the code itself, therefore any change to the code is reflected on the documentation. There is no need to maintain an external document.

```
@RestController
@RequestMapping("/users")
@Api(value = "users", description = "Users endpoints")
public class UserController {

  @Autowired
  private UserService userService;

  @GetMapping
  @ResponseStatus(HttpStatus.OK)
  public List<User> findAll() {
    return userService.findAll();
  }

  @PostMapping
  @ResponseStatus(HttpStatus.CREATED)
  public User create(@RequestBody UserDto userDto) {
    return userService.create(userDto);
  }

  @DeleteMapping
  @ResponseStatus(HttpStatus.NO_CONTENT)
  @RequestMapping(value = "/users/{userId}")
  public void remove(@PathVariable Long userId) {
    userService.removeById(userId);
  }

}
```

Listing 3.10: An example of Spring controller with Swagger's annotations that can be used to generate API documentation.

On the other hand, the push for clear, easy to read documentation has popularized the "design-first" approach. Not only more developers can have input on the documentation, but it actually results in cleaner code, because the developers are forced to think simpler, more concise, and easy to follow. The framework contains an editor that allows to write up the documentation in appropriate formats and have it automatically compared against the Swagger specification. Any mistakes are flagged, and alternatives are suggested. This way, when developers publish the documentation they can be sure that it's "error-free".

[6]Swagger is a specification, and supports a wide range of frameworks and their annotations.

As can be seen in the listing 3.11, the documentation consists of 2 parts, the operations and the models.

In either case the framework exposes the endpoints from the documentation controller and makes them accessible which allow for applications to be built upon the documentation. The application that will be developed in the thesis will take an advantage of the exposure. It will take an existing JSON document created by the framework and build an interactive documentation and testing environment upon it.

```
{
  "apiVersion": "1.0",
  "swaggerVersion": "1.0",
  "basePath": "http://localhost:8080",
  "resourcePath": "/users",
  "apis": [
    {
      "path": "/users",
      "description": "Users endpoints",
      "operations": [
        {
          "httpMethod": "GET",
          "summary": "findAll",
          "deprecated": "false",
          "responseClass": "List[User]",
        },

        ...
      ]
    }
  ],
  "models": [
    "User": {
      "properties": {
        "id": {
          "type": "long"
        },
        "firstname": {
          "type": "string"
        },
        "lastname": {
          "type": "string"
        }
      }
    }

    ...
  ]
}
```

Listing 3.11: An example of API documentation in JSON format created using the Swagger framework.

Chapter 4

Application Design

This chapter covers the requirements for the Restty application and presents the application's model and design drafts in detail. The model is designed using Entity Relationship Diagram which describes the entities and their relations, and may be useful for visualizing database design ideas, identifying the mistakes and design flaws before creating the database. The design drafts on the other hand, can help reveal any clashing visual elements while it is still easy to change them before writing the code. Moreover, the drafts allow to fully flesh out the ideas and choose the best possible options. The design drafts pictured in the following sections consists of separate UI elements that once combined, represents the final design draft of the application.

4.1 The Restty's Model

The application, as illustrated in the figure 4.1, consists of three major entities – projects, endpoints and test cases. It will allow users to create multiple projects to work on, with every project containing various endpoints and test cases. Each project, specified by its name and the Swagger's API file, draws its endpoints from that said file. Depending on the content of the file, endpoints may vary in headers (specified for the particular endpoint or for all of them), response statuses and parameters. The parameters may vary in types, for instance the endpoint can contain path variables, query parameters, or body data, which can be either primitive data types or objects that are represented by model entities.

In addition, the application contains test cases. Test cases consists of unspecified amount of endpoints that can have separate settings – custom values of parameters or models, for each test case. In addition, both the endpoint and test cases will contain logs of previous test runs, to provide users information about the tests that were executed in the past, and their results.

4.2 Requirements for the Restty Application

As mentioned earlier, the aim of the thesis is to create an application that allows to test endpoints of interfaces of other applications as well as to create extensive test cases from said interfaces. The interface that should be tested will be provided via the Swagger's JSON file which contains all necessary information about the interface. Therefore the Restty has to allow the users to choose the application's interface upon which the Restty will work. If the interface and its endpoints is successfully loaded and the data are persisted

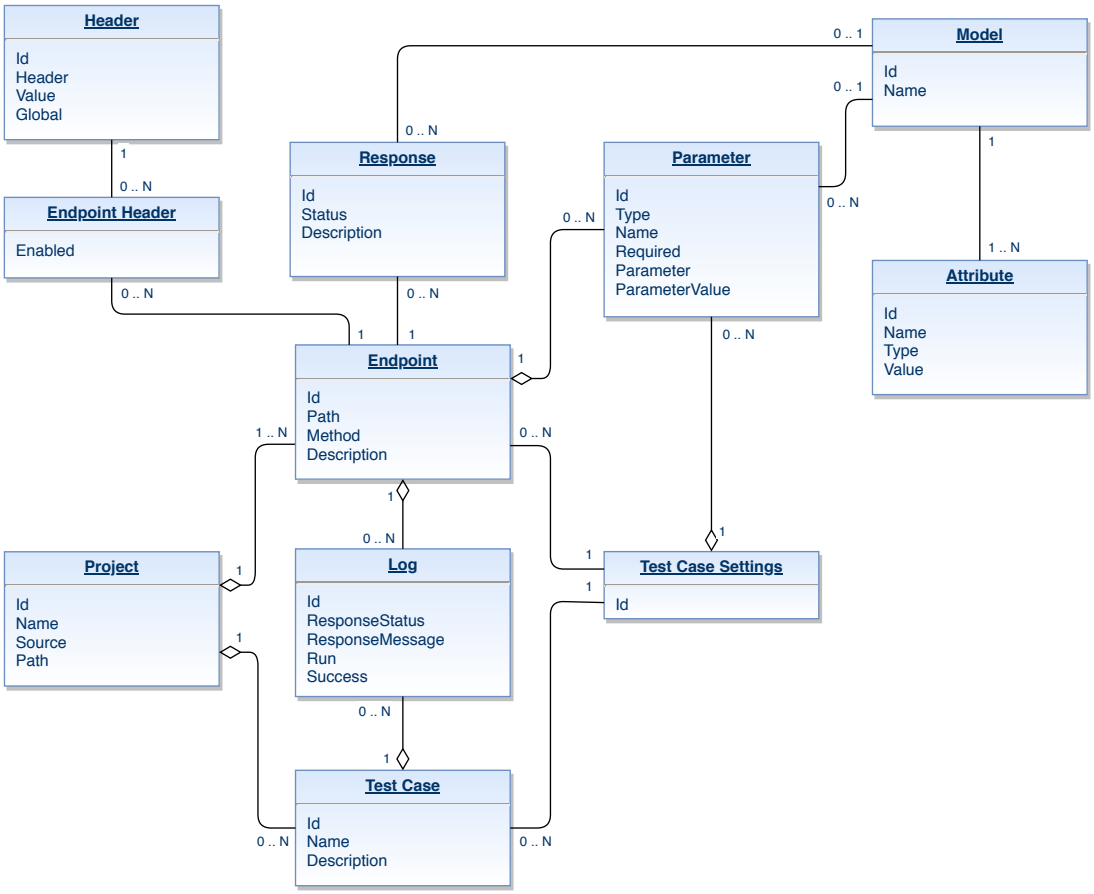

Figure 4.1: The ER Diagram that describes how the entities relate to each other within the application.

in the database, the user will be redirected to the dashboard which will display basic statistics about the current state of the endpoints and test cases.

From the dashboard the users will be able to navigate to the overview of all interface's endpoints, their details, configurations and past test runs. In addition, the users should be able to navigate to the similar overview with test cases, which the users can create using the Restty's Test Case Creator (TCC). The TCC should allow create test cases that can combine various endpoints. To clarify, it should be possible to test an API's endpoint and use its output as an input of another endpoint.

4.3 Design drafts

As mentioned earlier, the design drafts were created to reveal any clashing visual elements before writing the code. However, even though the thesis presents the final drafts of the application, it is still possible that parts of the design will be changed during the development process.

4.3.1 Designing the Project Explorer

The entry point of the application should be the Project Explorer. The Project Explorer, illustrated in the figure 4.2, should allow users to create and manage the projects created within the Restty. Each project should have a unique name which distinguishes it in the application. When the project is created by the user, besides its name, it needs to have specified a source of the project's endpoints. As a source (in form of a URL) is considered Swagger JSON with information about the project's interface. The source should be then passed to the Restty's backend that should make an request for the interface, parse the request's response and persist the information about interface's endpoints, its parameters, responses, and models, to the database. If the project information are successfully persisted in the Restty's database, the user should see the newly added project and its attributes, such as amount of endpoints and test cases, in the Project Explorer's list table.

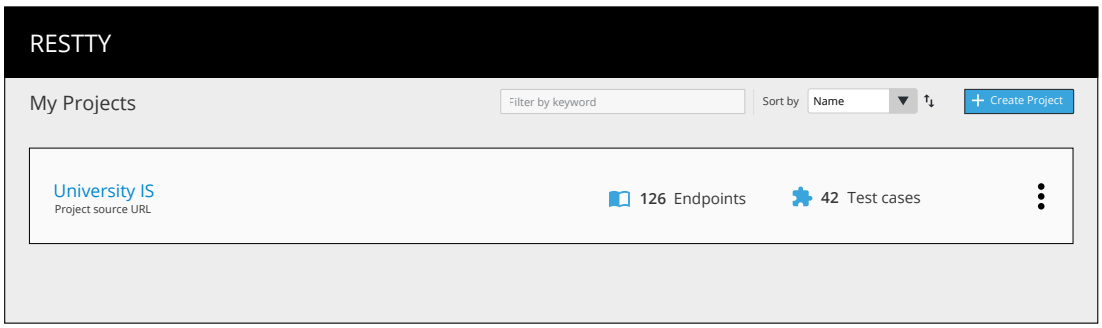

Figure 4.2: The Project Explorer that contains list of projects created in the Restty application.

Overall, the list table should show the users some basic information about their projects and should allow them to manage them. Finally, by clicking on the list item the users should be redirected to the specified project's dashboard.

4.3.2 The Project Dashboard

The dashboard is the entry point of each specific project. As can be seen in the figure 4.3, it consists of a horizontal and vertical navigation bars, donut charts and tables that contains latest information about the project. To keep things simple, the horizontal navigation bar should allow users to swiftly switch between projects without the need to access the Project Explorer. On the other hand the vertical navigation bar should allow users to quickly navigate between the project's endpoints, test cases and settings.

However, the main aim of the Dashboard is to show the current state of the project to the user. Which is achieved by showing the user two donut charts and two tables. The donut charts displays the state of the project's endpoints and test cases – how many of them were successfully (or not) tested, or if they were not tested at all. The tables on the other hand shows the user the information about the recent API or test runs. The Recent table shows the user last five runs no matter if they were successful or not. Whereas the Failures table shows the user unsuccessful runs only, if there are any. From there on, using the navigation bar or the quick navigation in the tables, users can start exploring the endpoints or the tests.

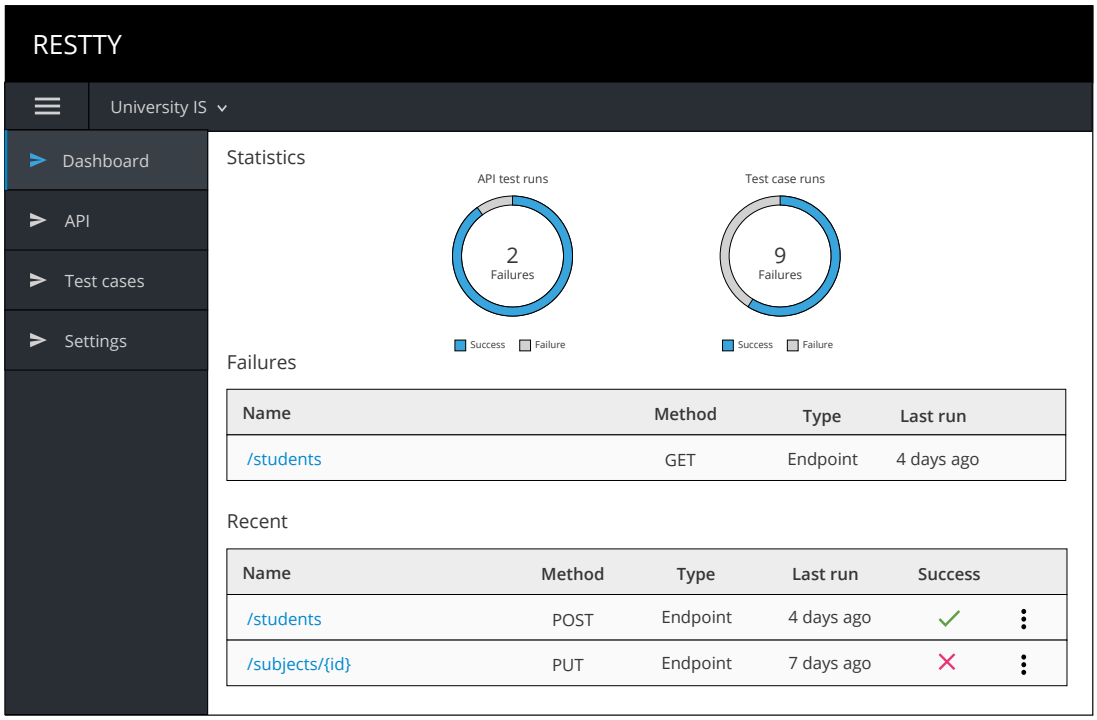

Figure 4.3: The dashboard of a project with lastest information about test runs.

4.3.3 Exploring the Endpoints

Once the users start working with the application, they should be able to view, manage and run the API endpoints they imported. However, as explained in the section 2.3, many applications which provide an overview of the endpoints does not scale well enough if they have to display a large number of the endpoints. To address this issue, the Restty puts the endpoints into filterable list table with a simple expansion for each item.

As illustrated in the figure 4.4, each item contains information about its last test run and its success. Moreover, the item's expansion contains the details of the item's particular endpoint such as request headers, path variables, parameters and responses.

In addition, when clicking on the endpoint's name, its detail is displayed. The detail consists of a tab view with three tabs. The first tab is similar to the list item's extension, showing the information about request and responses. The next one contains a configuration that allows to make changes to the endpoint e.g. changing the request params, body or headers. The last tab contains records of previous test runs, so that users know in which point the endpoint stopped working or when was the endpoint's code fixed.

4.3.4 The Test Cases

Even though it is important to enable users to work with the endpoints directly, the main focus of the Restty is on test cases. When users view the test cases page, they should be able to manage them. The management of the test cases is outlined using the table with filtering, sorting and pagination components. Moreover, each item in the table contains basic information about the specific test case (e.g. the date of last run, its success etc.).

Upon clicking on the test case's name, its detail is shown in a similar fashion as the endpoints detail. The view consists of a tab view, in which the most important tab is the con-

Figure 4.4: Expanded API endpoint in the endpoint's list table.

figuration tab – containing the Test Case Creator (TCC). The design of TCC is based on canvas, which is a HTML 5 element that allows for dynamic, scriptable rendering of 2D shapes and bitmap images. TCC was designed as an dynamic editor that allows users to build test cases using various endpoints an other test cases. The idea behind it is that the users will be able to construct complex test cases in the same way as they would create a linked list[1].

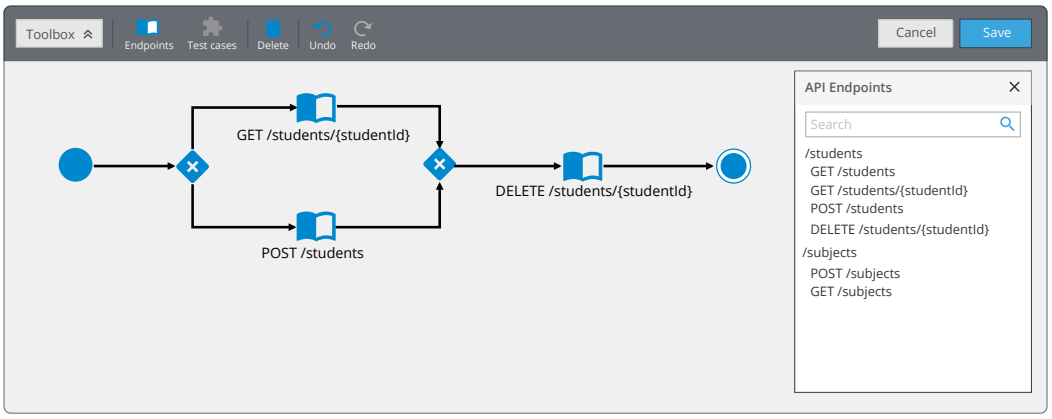

Figure 4.5: The TCC with an example of a test case.

The TCC consists of several parts. All of them together allow users to model desired test cases using the drag and drop functionality. To clarify how it works, consider an example from the figure 4.5. In the figure, using the drag and drop functionality was inserted several endpoints from the list to the canvas. When the endpoint is dropped, a modal window, as illustrated in the figure 4.6, with the endpoint's details appears. Note that the window is prefilled with the endpoint's details, loaded from its configuration.

However, it is important to notice that the behavior specified above is applied only to first endpoint that is added to the canvas. For the following endpoints, the input data is

[1] A linked list is a linear collection of data elements, in which linear order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a group of nodes which together represent a sequence.

automatically derived from the previous endpoint output. Once again, consider an example from the figure 4.5 that demonstrates how the data are passed bewteen the endpoints. In the beginning a student resource is found or created and afterwards immediately deleted, testing the deletion of the student resource.

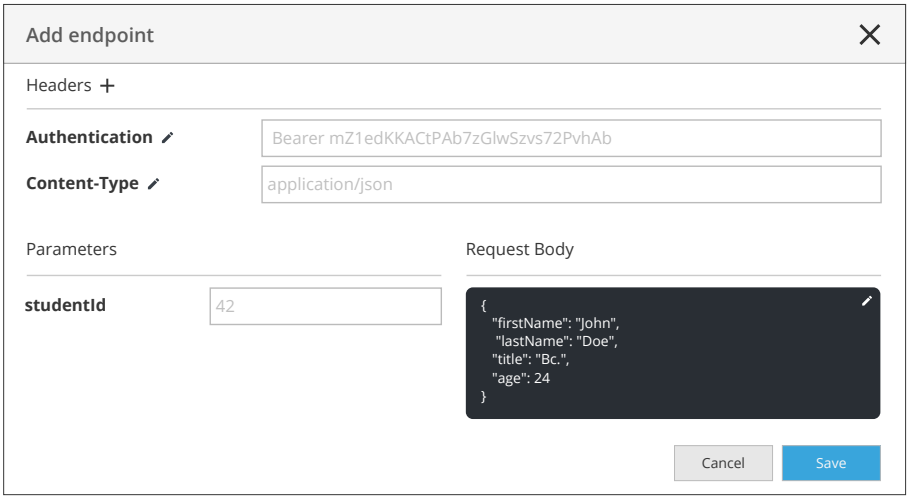

Figure 4.6: A modal window that appears when adding first endpoint to the canvas.

Chapter 5

Implementation and Testing

The chapter covers the implementation of the Restty application, based on the designs presented in the previous chapter, and its testing using the Red Hat JBoss BPM Suite application as a source of API endpoints to test. Backend of the application is implemented using Java 8, MVC framework Spring Boot, persistence framework Hibernate 5 and in memory database H2. The frontend is implemented using Angular 5, TypeScript language and the web framework PatternFly.

5.1 Separation of Concerns

Before diving into the implementation, it is important to specify the separation of concerns between the presentation layer (frontend), and the data access layer (backend) of the application. The frontend is primarily implemented using Angular while backend is implemented using Spring Boot. Combining these frameworks allows the creation of a minimal, but runnable, application with as little dependencies and setup as possible.

Therefore by following the separation of concerns principle, the application is split into separate Maven modules for the frontend and backend, where each module has its own POM[1] configuration file linked to the parent configuration file. To build the Angular application with Maven, the frontend's POM file have to contain the Frontend Maven Plugin dependency, which is needed to install prerequisities needed by the application and to keep the frontend and backend builds as separate as possible. However, even with the Frontend Maven Plugin the Angular application will not ended up in the final jar. To ensure that, the packaged Angular application have to be added to backend's resources. This whole approeach ensures the separation of concern principles and allows the whole application (frontend and backend) to be build using Maven's install command only.

5.2 Structure of the Frontend

When implementing the Restty, the most important task is to structure the application's code the right way. Ideally, the whole application should be a tree of components that implement clearly defined inputs and outputs, and minimize "two-way" databinding. That way, it is easier to predict when data changes and what the state of a component is.

[1] A Project Object Model (POM) is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project.

When developing Restty, the goal was to create an application that is scalable and consists of reusable, well encapsulated components that are easy to maintain and refactor. To follow these rules the application was divided into three parts, as can be seen in the figure 5.1.

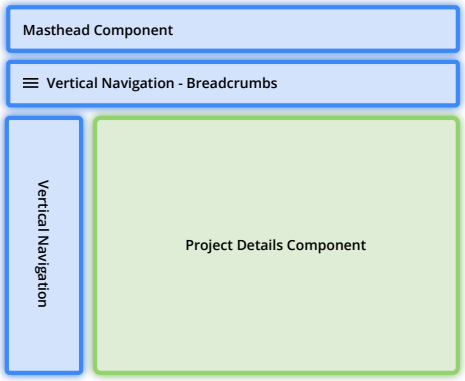

Figure 5.1: The structure of a Restty's main layout components.

The only component that is always present is the Masthead that serves as an app's header, containing the Restty's name. It is purposely made using the tall option of the PatternFly framework to accommodate for larger app's logo. Following the design, if the users open a project, the rest of the components appears, with one of them being the global navigation that is displayed on the left. While vertical navigation menus generally consume more space than their horizontal counterparts, they have become popular as desktop monitors move to "wide-screen" formats.

Moreover, vertical navigation supports common left to right flow and navigation categories are easily differentiated from other information that may exist in the header area of the application. Which in Restty's case contains a hamburger menu and a breadcrumbs navigation. The hamburger menu is always visible in the top left corner and allows to use the vertical navigation even on small devices using *@media* queries and PatternFly's *navbar-collapse* class, as can be seen in the listing 5.1.

```
@media (min-width: 768px) {
  .navbar-collapse.collapse {
    display: block !important;
    height: auto !important;
    padding-bottom: 0;
    overflow: visible !important;
  }
}

.collapse {
  display: none;
}
```

Listing 5.1: Media queries on the *.navbar-collapse* class that enables the navigation bar to work even on small devices.

Breadcrumbs displays a users location within an application hierarchy and act as a resource to help users navigate more efficiently and provide additional context. Note that the breadcrumbs originally were not part of the designs. They replaced a context selector

that should allow for quick changing between various projects, but it was decided that the breadcrumbs are more useful to the user as the secondary navigation items like tables are not always exposed.

The last component in the layout is the *ProjectDetailComponent* that creates a container for all subpages, such as endpoint and test cases tables, details, etc. As can be seen in the listing 5.2, the subpages are displayed using routing which is a technique that can interpret a browser URL as an instruction to navigate to a client generated view.

```
// Handles navigation bar clicks
onItemClicked($event: NavigationItemConfig): void {
  this.contentView = $event.title;
  if ($event.title === 'Dashboard') {
    this.contentView = null;
    this.router.navigate(['projects', this.project.id]);
    } else if ($event.title === 'API') {
      this.router.navigate(['projects', this.project.id, 'api']);
    } else if ($event.title === 'Test cases') {
      this.router.navigate(['projects', this.project.id, 'test-cases']);
    } else if ($event.title === 'Settings') {
      this.router.navigate(['projects', this.project.id, 'settings']);
    }
}
```

Listing 5.2: Solution of routing the subpages in the *ProjectDetailComponent*.

5.3 Parsing the Swagger's JSON

When creating the project within the Restty, the application consumes Swagger's API file and parses the information about project's interface from it. As can be seen in the listing 3.11 the content of the file may be extremely large and complicated.

Initially, when creating the project, the source of the file is passed to the backend, in which the *createProject(ProjectDto projectDto)* method in the implementation of *ProjectService* handles the project creation. At first, the content of Swagger's file is fetched using the Spring's *RestTemplate* class and then it is parsed to Data Transfer Objects (DTO) of which, the database entities are created.

```
ObjectMapper mapper = new ObjectMapper(_;
JsonNode rootNode = mapper.readTree(json);

String scheme = JsonUtils.getScheme(rootNode);
String host = JsonUtils.getPathValue(rootNode, HOST_PROPERTY, false);
String basePath = Objects.toString(JsonUtils.getPathValue(rootNode, "basePath", false), "");
if (StringUtils.isBlank(scheme) || StringUtils.isBlank(host)) {
  this.basePath = source.substring(0, source.lastIndexOf("/") + 1) + basePath;
} else {
  this.basePath = scheme + host + basePath;
}
```

Listing 5.3: Parsing the address of the test server from Swagger's API file.

The file contains various information about the project and its API, but for the Restty's purposes it is important to parse only the information about the test server, endpoints and the models. The information about test server is stored inside the *host*, *basePath* and *schemes* attributes which, once combined, creates a complete path that will be used

as default when making API calls. Note that according to Swagger's specification, the *host* and *scheme* can be omitted for a more dynamic association. In that case, the host and scheme used to serve the API documentation is used for API calls.

Once the information about the test server are parsed, the application proceeds to parsing the data for the entities for which it needs to process two attributes – *paths* and *definitions*. The paths consists of list of objects where each object represents particular endpoint. In addition, each endpoint contains objects represented by HTTP methods that are allowed for the particular endpoint. Finally, each HTTP method object contains the detailed information about the endpoint like its description, parameters or responses. On the other hand, the definitions consist of a list of objects that represents entities used within the imported project's API. These definitions are then referenced from the endpoints as parameters or response bodies.

```
ObjectNode definitionsNode = JsonUtils.getObjectNode(rootNode, "definitions", false);
 if (definitionsNode != null) {
  definitionsNode.fields().forEachRemaining(definitionNode -> {
    ModelDto model = new ModelDto();
    model.setName(definitionNode.getKey());

    ObjectNode propertiesNodes =
      JsonUtils.getObjectNode(definitionNode.getValue(), "properties", false);
    if (propertiesNodes != null) {
      propertiesNodes.fields().forEachRemaining(propertyNode -> {
        AttributeDto attribute = new AttributeDto();
              attribute.setName(propertyNode.getKey());

              String type =
                JsonUtils.getPathValue(propertyNode.getValue(), "type", false);
              if (StringUtils.isNotBlank(type)) {
                attribute.setType(type);
                  model.addAttribute(attribute);
              } else {
                String modelName =
                  JsonUtils.getPathValue(propertyNode.getValue(), "ref", false);
                  if (StringUtils.isNotBlank(modelName)) {
                    attribute.setType(modelName.substring(modelName.lastIndexOf('/') + 1));
                      model.addAttribute(attribute);
                  }
              }
          });
        }

        models.add(model);
    });
 }
```

Listing 5.4: Parsing the API's model definitions from the Swagger's API file.

5.4 Implementation of the API Testing

This section contains the details about implementation of the cornerstone of the whole application – how the test cases are created and how they are, along with endpoints, run. By default, users can run the endpoints straight away but they most likely will not work. That is because the application does not know the parameter values required

by the endpoints. Therefore users are needed to specify the parameters of endpoints to be used in future endpoints or test cases runs.

```
<div class="flex-info-left request-params bold">
  <app-edit-param (paramEditEvent)="refreshOnParamEdit($event)" [parameter]="parameter">
  </app-edit-param>
  <div class="edit-parameter inline-block">
    <span class="pf pficon-edit clickable" (click)="createModal('#editParamModal', parameter.id)">
    </span>
  </div>
  <div class="inline-block">
    <span>{{parameter.name}}</span><br>
    <span class="font-size-sm">(query)</span>
  </div>
</div>
```

Listing 5.5: Frontend's implementation of setting the parameter's value in which all logic is handled by the matching TypeScript file.

Once all parameters required by the endpoint are specified, the users may run the test using appropriate button. The button is connected to a listener, which, on click, triggers a REST API call to the backend with the id of the endpoint that should be tested. On the backend, the request is handled by the *EndpointController* that finds all available information about said endpoint and, as can be seen in the listing 5.6, passes it to the appropriate service that handles the test.

```
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
endpoint.getHeaders().stream().filter(header -> header.getEnabled()).forEach(endpointHeader -> {
  headers.add(endpointHeader.getHeader().getHeader(), endpointHeader.getHeader().getValue());
});

List<Parameter> pathVariables = endpoint.getParameters()
  .stream()
    .filter(pathVariable -> ParamType.PATH.equals(pathVariable.getType()))
    .collect(Collectors.toList());

String path = endpoint.getProject().getPath() + endpoint.getPath();
String resolvedPath = resolvePathVariables(path, pathVariables);

Optional<Parameter> body = endpoint.getParameters()
  .stream()
  .filter(parameter -> ParamType.BODY.equals(parameter.getType()))
  .findFirst();

JSONObject jsonBody = new JSONObject();
if (body.isPresent()) {
  body.get().getModel().getAttributes().forEach(attr -> {
    jsonBody.put(attr.getName(), attr.getValue());
  }

HttpEntity<String> entity = new HttpEntity<>(jsonBody.toString(), headers);
restTemplate.exchange(resolvedPath, endpoint.getMethod(), entity, String.class);
```

Listing 5.6: The implementation of a test run in backend's service.

The test is implemented using Spring's *RestTemplate* class which allows calling customizable HTTP requests from the backend services. Furthermore, when the test is ex-

ecuted, the logs are created for the endpoint or test case that was subject of the test, keeping the information about the run persisted. Finally, the testing of test cases is implemented in a similar fashion. That is because in fact, a test case is more or less a linked list of endpoints.

However, there is a slight difference in the test cases implementation. Obviously, the users might want to use different parameter values for the same endpoint in different test cases which is not possible with the current implementation of the endpoint's parameters. Therefore the *TextCaseSettings* class was created. It links the endpoint with specific test case and copies the endpoint's parameters to its own properties which means that by default, if the endpoint is added to the test case, it contains the parameters that were specified in the endpoint's configuration. However, the parameter values in the *TestCaseSettings* class may be modified without it having an effect on the original endpoint's values. Overall, this approach allows the users to create test cases with specific conditions which results in better test coverage of the tested project's interface.

5.5 Testing with Red Hat JBoss BPM Suite

The testing of the application consisted of two parts. The first part consisted of testing the Restty's REST API interface using the cURL and Postman applications which validated that the backend of the Restty works as expected. The second, more important part, consisted of the manual testing of the application as a whole using the Red Hat JBoss BPM Suite application's remote interface.

5.5.1 Red Hat JBoss BPM Suite's Basic Concepts

The Red Hat JBoss BPM Suite is an open source business process management suite that combines Business Process Management and Businnes Rules Management and enables business and IT users to create, manage, validate, and deploy Business Processes and Rules. To accomodate Business Rules component, JBoss BPM Suite includes integrated Red Hat JBoss BRMS which is a comprehensive bussiness automation platform for bussiness rules management, business resource optimization, and complex event processing.

The Red Hat JBoss BRMS and Red Hat JBoss BPM Suite use a centralized repository where all resources are stored. This ensures consistency, transparency, and the ability to audit across the business. Business users can modify business logic and business processes without requiring assistance from IT personnel.

The application provides tools for creating, editing, running, and runtime management of BPMN process models. The models are defined using the BPMN2 language, but more importantly, they can be created using the JBoss BPM Suite API which will be subject of the testing. To briefly point out options of what BPM Suite can do, it is necessary to provide a bit of background on what rules, events and processes are.

A business process is a process that describes the order in which a series of steps need to be executed, using a flow chart. Next, the events specify moments in the execution process. The events are fired by the BPM's engine during graph execution. An event is always relative to an element in the process definition. Last but not least, a business rule is a rule that defines or contrains some aspect of process and always resolves to either true or false. In other words, rules are linked with the process models to enforce the correct policies at each process step.

5.5.2 Testing the Restty

As mentioned earlier, the testing was divided into two parts. The first part was focused on testing Restty's REST API using cURL at first and Postman later for more complicated API calls. The API consists of several endpoints which are managed by appropriate controllers on the backend. In this section, the focus will be on testing the *ProjectController* that handles all endpoints, in which the path starts with */api/projects*.

When starting the application, the users will come in first contact with projects dashboard e.g. it was needed to create an endpoint that would find all existing projects in the Restty application. In the listing 5.7 can be seen the code used for testing the said endpoint, returning the list of all projects within the application.

```
curl http://localhost:8081/api/projects
[
  {
    "id": 20,
    "name": "swagger-models",
    "source": "http://petstore.swagger.io/v2/swagger.json",
    "tests": 1,
    "endpoints": 20
  }
]
```

Listing 5.7: Testing the */api/projects* endpoint using cURL.

As can be seen in the listing 5.7, the project with BPM Suite's API is missing, therefore it is needed to create it using the same endpoint as for finding the projects within the Restty, but using the POST method. In the listing 5.8 can be seen an example of such API call.

```
curl --request POST \
  --url http://localhost:8081/api/projects \
  -d '{
    "name": "bpm-suite", \
    "source": "http://localhost:8080/kie-server/services/rest/server/swagger.json" \
  }'
```

Listing 5.8: Testing the creating of an project within the Restty.

Once the BPM Suite's project is created, it is possible to step up in the testing process, meaning to test the application as a whole using remote interface provided by the project. In particular, for testing purposes was imported BPM's process server execution module which allows users to communicate with the process server through REST API. The first step might be to test simple GET request, for instance the */server/containers* which retrieves containers deployed on the test server. After the test is completed, the user is informed about its result using a PatternFly's notification. This process may be repeated multiple times for multiple endpoints furthermore testing the application.

However, as was stated earlier, the functionality to test separate API endpoints is provided by many other applications. Therefore it is necessary to properly test the functionality of the test cases. Consider a following test case – the user wants to test a deletion of a process instance. Normally he would have to create the instance manually, retrieve its id and then delete it using the id. However, using Restty, the process can be automatized. User may simple create a test case and to that test case insert two API requests. The first request would be the POST request with parameters filled either from endpoint's configuration or by the user and the following request would be DELETE request that would use

the parameters from the previous request. After running the test, the user is again informed about its outcome and most importantly, he does not have to modify the test if he would want to run it again at some point in the future.

5.6 Future extensions

Even though the Restty makes testing much easier for developers, it does not reach its full potential within the thesis. The testing is an extensive discipline therefore the application has always room for improvements. To make it even more usable in practice, the following extensions were suggested.

1. The logs that stores the information about previous test runs could contain more information about the run besides the response code and response message. It is possible to save the exact format of the request, its headers, parameters, etc., and of the response.

2. The Project Explorer might offer users to export existing projects or to import projects that were previously created.

3. The Restty could offer a migration to the newer version of the API, for instance on the backend could be implemented a cron job that would check if the Swagger's API file has changed and if it has it could offer an automatic migration.

4. Finally, the Restty could create trivial test cases on its own – for instance, the tests for deleting a resource or testing the pagination etc., could be done automatically, saving time for the developers.

5. The test cases can be extended to allow users create conditions. For instance, the test case could contain a request that would tried to find a requested resource. If the resource does not exist, it could be created by using another appropriate request.

Obviously, the extensions stated above are not the only things that can be added to the application. It is always possible to optimize, the application itself both in terms of performance and user experience.

Chapter 6

Conclusion

The aim of the thesis was to design and develop an application that allows its users to test API endpoints of other applications and to create extensive test cases from said endpoints. As a part of working on the above goal, I studied the topics of Web Services, in particular the RESTful Web Services and their application programming interfaces. On top of that I studied the technologies needed for the development of the application and conducted a research on existing API testing solutions.

In Chapter 4, I designed a custom solution of the problem using the design integrity of the PatternFly framework. The designs, that were several times consulted in Red Hat, were used to reveal any clashing visual elements or design flaws before writing the code. After several reworks and improvements they were used as a basis for the development of the application.

The development consisted of building the client side (frontend) and the server side (backend) of the application as separate sections, following the Separation of Concerns principles. The achievement is that the backend of the application is fully capable of parsing the Swagger's API file, running and managing its endpoints, and allowing to create extensive test cases from said endpoints. In addition, the frontend of the application provides clear and easy to use interface, even in case of large remote interfaces.

Finally, as stated in the section 5.5, the developed application – Restty, was tested on Red Hat JBoss BPM Suite application, which provided a large remote interface consisting of various different endpoints that were used to reveal any errors or bugs in the application and to learn more about the Restty's future improvements, such as automatic test case creation or automatic migrations to the newer versions of the remote interfaces. Overall, the application serves its purpose and has a potential to become the leading application for the API testing processes and test automation.

Bibliography

[1] Berners-Lee, T.: Uniform Resource Identifiers (URI): Generic Syntax. August 1998. [Online; visited 20.01.2018].
Retrieved from: https://www.ietf.org/rfc/rfc2396

[2] Dewailly, L.: *Building a RESTful Web Service with Spring*. Packt Publishing. 2015. ISBN 978-1785285714.

[3] Fielding, R. T.: Architectural Styles and the Design of Network-based Software Architectures [dissertation]. University of California, Irvine. 2000.

[4] Fielding, R. T.; et al.: Hypertext Transfer Protocol – HTTP/1.1. June 1999. [Online; visited 10.12.2017].
Retrieved from: https://tools.ietf.org/html/rfc2616

[5] Frisbie, M.: *Angular 2 Cookbook*. Packt Publishing. 2017. ISBN 978-1785881923.

[6] Hevery, M.: Hello World, <angular/> is here. September 2009. [Online; visited 17.12.2017].
Retrieved from:
http://misko.hevery.com/2009/09/28/hello-world-angular-is-here/

[7] James Gosling, G. L. S. J., Bill Joy: *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional. 2014. ISBN 978-0133900699.

[8] Leonard Richardson, S. R., Mike Amundsen: *RESTful Web APIs: Services for a Changing World*. O'Reilly Media. 2013. ISBN 978-1449358068.

[9] Mihalcea, V.: *High-Performance Java Persistence*. VLAD MIHALCEA. 2016. ISBN 978-9730228236.

[10] Richardson, A. J.: *Automating and Testing a REST API: A Case Study in API testing using: Java, REST Assured, Postman, Tracks, cURL and HTTP Proxies*. Compendium Developments Ltd. 2017. ISBN 978-0956733290.

[11] SmartBear Software: Swagger, the world's most popular API tooling. [Online; visited 10.12.2017].
Retrieved from: https://swagger.io/

[12] Spurlock, J.: *Bootstrap: Responsive Web Development*. O'Reilly Media. 2013. ISBN 978-1449343910.

[13] Stenberg, D.: cURL: Command line tool and library. 1997. [Online; visited 13.02.2018].
Retrieved from: https://curl.haxx.se/

[14] W3C: Web Services Glossary § Web Service. February 2004. [Online; visited 10.12.2017].
Retrieved from:
https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice

Appendix A

Installation details

The Chapter covers the details of running the Restty application on local computer. At first it is required to copy the folder with the project to the local disk. Then it is neccessary to follow the procedure that consists of two parts. The first part is used to start up the backend of the application and the steps needed to run Restty's backend are as follows:

1. Download and install the Java Development Kit of version 1.8.
2. Download and install the Apache Maven project management tool in version 3 or later.
3. Add the Java and Maven to the system variables if running the application on Windows.
4. In the root folder of the project run the command – *mvn clean install* which installs neccessary dependencies required by the application.
5. To start the application, in the */backend* folder, run the command – *mvn spring-boot:run*.

The second part is to start up the frontend of the application which is needed for Restty to be fully functional. To start up the frontend, following steps must be executed:

1. Download latest version of NodeJS.
2. Install the *angular-cli* using the command – *npm install -g @angular-cli*.
3. To start the application, run the *ng serve –proxy-conf proxy.conf.json* command in the */frontend/src/main/frontend* directory.