



**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**

NÁZEV PRÁCE

**TERM PROJECT**

SEMESTRÁLNÍ PROJEKT

**AUTHOR**

AUTOR PRÁCE

**Bc. ONDŘEJ KRPEC**

**SUPERVISOR**

VEDOUCÍ PRÁCE

**Ing. RADEK KOČÍ, Ph.D.**

**BRNO 2017**

## Abstract

The thesis describes a draft of an application based on Swagger Framework that will allow to create test cases from the application's API. The theoretical part of the thesis explains the principles of web services, remote interfaces, technologies that will be used for development, and the Swagger Framework itself. Subsequently, the design drafts of the application are presented in detail. The results of the thesis allow development of the application that will provide an option to create test cases through calling web services that are provided by the APIs. This will lead to decrease need of funds for development and for the simpler realization of continuous testing.

## Abstrakt

Tato práce popisuje návrh aplikace, která na základě nástroje Swagger umožní vytvářet testovací scénáře z API aplikací. Teoretická část práce vysvětluje principy webových služeb, vzdálených rozhraní a představuje nástroj Swagger. V práci jsou dále prezentovány designové návrhy aplikace a jsou zde představeny technologie, které budou použité při jejím vývoji. Výsledky této práce umožní vytvořit aplikaci, která poskytne možnost testovat API aplikací a navíc na základě těchto API vytvářet testovací scénáře – kombinující jednotlivé části API, a tím snížit náklady na vývoj aplikací a umožnit realizaci průběžného testování.

## Keywords

REST, Continuous testing, Swagger, PatternFly, Angular, Red Hat, Automated testing, Web Services, Test cases

## Klíčová slova

Webové služby, REST, Automatické testování, Průběžné testování, Swagger, PatternFly, Angular, Red Hat, Testovací scénáře

## Reference

KRPEC, Ondřej. *Remote API Web Reference for Java Enterprise Applications*. Brno, 2017. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Kočí, Ph.D.

# Remote API Web Reference for Java Enterprise Applications

## Declaration

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

.....

Ondřej Krpec  
January 8, 2018

## Acknowledgements

I would like to thank Mr. Mrg. Ivo Bek for his technical leading of this Master Thesis. At the same time, I would like to thank Mr. Ing. Radek Kočí, Ph.D., for his pedagogical leadership.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>2</b>  |
| <b>2</b> | <b>Preliminaries and Definitions</b>                         | <b>3</b>  |
| 2.1      | Understanding the Web Services . . . . .                     | 3         |
| 2.1.1    | Introduction to RESTful Web Services . . . . .               | 3         |
| 2.1.2    | Messaging . . . . .  | 4         |
| 2.1.3    | Addressing the Resources . . . . .                           | 6         |
| 2.1.4    | Representation of the Resources . . . . .                    | 7         |
| 2.2      | Introduction to Application Programming Interfaces . . . . . | 8         |
| 2.2.1    | When is API RESTful? . . . . .                               | 8         |
| 2.2.2    | The Importance of Testing APIs . . . . .                     | 9         |
| 2.2.3    | The Frameworks for Testing APIs . . . . .                    | 10        |
| <b>3</b> | <b>Technologies and Frameworks</b>                           | <b>11</b> |
| 3.1      | Introduction to Swagger Framework . . . . .                  | 11        |
| 3.1.1    | Generating the API Documentation . . . . .                   | 11        |
| 3.2      | What is Angular? . . . . .                                   | 13        |
| 3.2.1    | Beginning as AngularJS . . . . .                             | 13        |
| 3.2.2    | Angular's Core Concepts . . . . .                            | 14        |
| 3.3      | Introducing TypeScript . . . . .                             | 16        |
| 3.3.1    | Why Add Types to JavaScript . . . . .                        | 16        |
| 3.3.2    | Future JavaScript . . . . .                                  | 17        |
| 3.4      | Styling with PatternFly . . . . .                            | 18        |
| 3.4.1    | Using the Components . . . . .                               | 18        |
| 3.4.2    | Working with the Grid . . . . .                              | 19        |
| <b>4</b> | <b>Application Design</b>                                    | <b>20</b> |
| 4.1      | Application's Requirements . . . . .                         | 20        |
| 4.2      | Designing the Project Explorer . . . . .                     | 20        |
| 4.3      | Showing the Endpoints . . . . .                              | 21        |
| 4.4      | Creating the Test Case . . . . .                             | 22        |
| <b>5</b> | <b>Conclusion</b>  | <b>24</b> |
|          | <b>Bibliography</b>  | <b>25</b> |
| <b>A</b> | <b>The Design Drafts</b>                                     | <b>26</b> |

# Chapter 1

## Introduction

In the software industry, the great, accessible and testable code is crucial for modern business, and the best way for us to test and share it is through APIs<sup>1</sup>. They are supposed to connect engineers, allow for sharing of developments, let companies add value to the products and create an ecosystem of shared knowledge. To fulfill these tasks they have to be clear, accessible and most importantly human and machine readable. However, despite their importance, there hasn't been an industry standard for documenting nor testing them.

This thesis aims to solve the problem of documenting and testing the APIs. The goal is to design and develop an application having an innovative user interface with regard to clarity and simple use, aimed to developers, even in case of large interfaces. The application will be based on Swagger framework that provides a way to automate API generation. The application will provide not only API reference, 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 are going to be used for implementation of the application. Chapter 4 describes the application design, wireframes and mockups. The last Chapter contains an overall summary of the designed solution and final thoughts on the work done within the thesis.

---

<sup>1</sup>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 sections, is introduced the basic terminology and established notion of remote interfaces and web services. In the next section is provided an explanation what APIs are, and why is important to automate their testing. The last section covers the introduction of a Swagger framework [6] and its importance to the thesis.

### 2.1 Understanding the Web Services

A web service is a software system designed to support interoperable machine to machine interaction 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 computer networks like the Internet in a manner similar to interprocess communication on a single computer.

In the past, web services used mostly SOAP<sup>1</sup> over HTTP protocol, allowing less costly interactions over the Internet. In a 2004, the W3C extended the definition of web services about “REST-compliant” web services [8], 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 stands for Representational State Transfer, which is an architectural style for networked hypermedia applications, primarily used to build web services. The term was first introduced and defined in the year 2000 by Roy Fielding in his doctoral dissertation [1]. Fielding’s dissertation explained 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 (URI) standards.

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

---

<sup>1</sup>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, while JSON being currently the most popular format being used.

### 2.1.2 Messaging

As mentioned in the previous section, RESTful web services can use any stateless communication protocol as a medium of communication between client and server. However, the HTTP protocol [2] is the most popular. A client sends a message in form of a HTTP Request and the server responds in the form of HTTP Response. This technique is termed as Messaging. Apart from the actual data, these messages also contain some metadata about the message itself.

As shown in figure 2.1 a request message from a client to a server contains of five major parts.



Figure 2.1: HTTP Request format

- **Verb** indicates the HTTP method like GET, PUT, POST, DELETE, etc.
- **URI** is the Uniform Resource Identifier used to identify the resource on the server.
- **HTTP Version** is the version of HTTP.
- **Request Header** contains the metadata as a collection of “key-value” pairs of headers and their values. For example, client (or browser) type, format supported by the client, format of the message body, cache settings for the response, and much more information.
- **Request Body** is the actual message content or resource representation.

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

```
1 GET / HTTP/1.1
2 Host: www.fit.vutbr.cz
3 User-Agent: Mozilla/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: A sample of a simplified GET Request.

The GET command is followed by the URI and the HTTP version. The request also contains some headers. The “User-Agent” header contains information about the type of a client which made the request. The “Accept” headers tells the server about the 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 a response at runtime depending on the value of the “Accept” header. Because the request in listing 2.1 is a GET request, it does not contain any message body.

When the server receives the request it should respond with a HTTP Response. The response has four major parts.

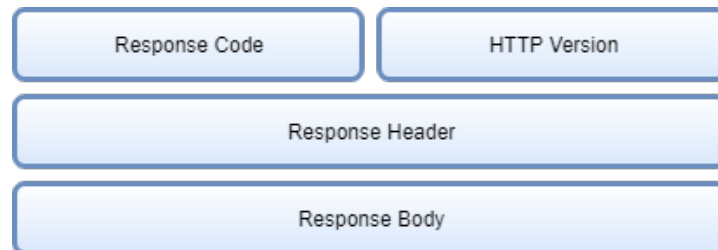


Figure 2.2: HTTP Response format

- **Response Code** contains the server status for the requested resource. This response is generally the “3-digit” HTTP status code. For example, 404 means resource not found and 200 means response is ok.
- **HTTP Version** is the version of HTTP.
- **Response Header** contains the metadata and settings about the response message as “key-value” pairs. For example, content length, content type, response date, etc.
- **Response Body** contains message content or resource representation if the request was successful.

In the listing 2.2, can be seen an example of the response to the GET request to Faculty of Information Technology website was made earlier.

```

1 HTTP/1.1 200 OK
2 Date: Sat, 09 Dec 2017 08:36:01 GMT
3 Server: Apache
4 Content-Location: index.php.cz
5 Pragma: no-cache
6 Keep-Alive: timeout=60, max=100
7 Connection: Keep-Alive
8 Transfer-Encoding: chunked
9 Content-Type: text/html; charset=iso-8859-2
10 Content-Language: cs
11 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
12 <html>
13 <head>
14 ...

```

Listing 2.2: A sample of a simplified response to a GET Request.

The response code *200 OK* means that everything is ok and the response message body contains a valid representation of the requested resource. In this case, the representation is an HTML document that is declared by “Content-Type” header in the response header. The other attributes in the header are self explanatory.



### 2.1.3 Addressing the Resources

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

$$< protocol > : // < serviceName > / < resourceType > / < resourceID >$$

The URI should not say anything about the operation or action. This enables us to call the same URI with different HTTP verbs to perform different operations. Verbs are used to identify the operation to be performed on the resource.

While designing a URI some important points should be considered.

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

To avoid confusion, let us have an example. Suppose a database of users and that the users should be exposed to the Internet through a service. We already know that the resources should be addressed by the URI and the action performed on the resource should be defined by a HTTP verb. The verbs correspond to read, create, update, and delete CRUD<sup>2</sup> operations.

#### Creating a user – POST /users

To create new resources, in this case a user, the POST verb is most often utilized. On successful creation, returning a “Location” header with a link to the newly created resource with the *201 Created* HTTP status.

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 the same information.

#### Retrieving a user – GET /users/{id}

The HTTP GET method is used to retrieve a representation of a resource, in this case a user with specified id. If the request is successful, it returns the user representation in XML or JSON format, like the one we can see in listing 2.3, respectively 2.4, and a HTTP response code of *200 OK*.

The GET requests should be used only to read data and not change it. Therefore, 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.

---

<sup>2</sup>In computer programming, create, read, update, and delete (as an acronym CRUD) are the four basic functions of persistent storage.

### Updating a user – PUT /users/{id}

For update capabilities, the PUT verb is often most utilized, “PUT-ing” to a known resource, in our case a user, with the request body containing the newly updated representation of the original resource. On successful update, the server should return *200 OK* or *204 No Content* 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.

PUT is not a safe operation, in that it modifies state on the server, but is idempotent. In other words, if we update the user using the PUT method and then make that same call again, the resource is still there and has the same state as it did with the first call.

### Deleting a user – DELETE /users/{id}

The DELETE verb is pretty straightforward, it is used to delete a resource, in our case the user with specified id. On successful deletion, the server should return HTTP status *204 No Content* with no response body.

“HTTP-spec-wise”, DELETE operations are idempotent. If we delete a user, it is removed. Repeatedly calling the method on that user ends up the same – the user is gone. There is a caveat about the method idempotence, however. Calling it on a resource a second time will result in *404 Not Found* status code since it was already removed and therefore is no longer findable. It makes the operations in fact no longer idempotent, however, the “end-state” of the resource is the same.

#### 2.1.4 Representation of the Resources

It is clear that the focus of a RESTful service is on resources and how to provide access to them. A resource can easily be thought of as an object as in OOP<sup>3</sup>. These resources can be text files, HTML pages, images, or videos, and can consist of other resources. The server simply provides access to resources and client accesses and modifies them. The architecture does not put a restriction on the format of a resource representation. However, as mentioned before, the most popular representations of resources are XML and JSON.

```
1 <user>
2   <id>1</id>
3   <firstName>John</firstName>
4   <lastName>Doe</lastName>
5   <age>42</age>
6 </user>
```

Listing 2.3: 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 client can understand the same format.

Let us consider an example of a user object. Its XML and JSON representation can look like the one in the listing 2.3 and 2.4, respectively.

---

<sup>3</sup>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.

```

1 {
2   "id": 1,
3   "firstName": "John",
4   "lastName": "Doe",
5   "age": 42
6 }

```

Listing 2.4: 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 a 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.

The API [5] approach creates a loosely coupled architecture that allows a component service to have a wide range of future uses, and it 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:

- **Automation.** With APIs, computers rather than people can manage the work. Through them, companies can update work flows to make them quicker and more productive.
- **Integration.** They allow content to be embedded from any site or application more easily. This guarantees more fluid information delivery and an integrated user experience.
- **Personalization.** Any user or company can customize the content and services that they use the most.
- **Reduction of costs.** 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.
- **Increasing customer loyalty.** 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?

In the previous sections we introduced the term Application Programming Interface and explained the RESTful web services. However, it is necessary to point out that not all 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.

### 2.2.2 The Importance of Testing APIs

API testing is testing that APIs and the integrations they enable work in the most optimal manner. This form of testing concentrates on using software to make API calls in order to receive an output before observing and logging the system’s response. Most importantly, this tests that the API returns a correct response or output under varying conditions. The output is typically as follows:

- A Pass or Fail status
- Date or information
- A call to another API

However there also could be no output at all or something completely unpredicted can occur. It makes the testing a crucial part of the application development process.

### 2.2.3 The Frameworks for Testing APIs

As APIs are becoming an integral part of how software works, and the more the users rely on “web-based” systems, the more crucial is that they are tested. Therefore it is not a surprise that there are many frameworks and applications that allow to test the APIs.

The simplest is the cURL<sup>4</sup>. It is a command line tool for transferring data using various protocols. It consists of two products, libcurl and cURL, where libcurl is a free “client-side” URL transfer library and cURL is the actual tool for getting and sending files using URL syntax. Together they allow testing the APIs using HTTP requests. The basic use of cURL involves simply typing *curl* at the command line, followed by the URL of the output to retrieve. As we can see in the listing 2.5, writing tests with cURL is not very intuitive. Therefore, other frameworks and applications, like the Postman, were developed.

```
1 curl -d '{
2   "firstName": "John",
3   "lastName": "Doe",
4   "age": 42
5 }' -X POST http://localhost:8080/users
```

Listing 2.5: An example of POST request that creates new user on the server using cURL.

Postman is a software that allows executing and managing HTTP requests. It offers a “user-friendly” interface with which is possible to make the requests without the hassle of writing code just to test an API’s functionality. Using the user interface, the developers can simply define request’s address, query parameters, headers, authorization methods and data, and even write simple scripts that are executed before or after running the request. It even supports an import of API definitions from the API documentation framework Swagger.

However, as can be seen, both of these applications have a serious drawback. What if, a developer wants to create test case that would execute first request and then took the response from the server and used it as a input for the second request? Using cURL this would mean writing additional code that would connect two requests and in the Postman application, this is not possible at all. Although Postman offers to concatenate the requests into a collection, it does not offer to use one’s request output as other’s request input.

---

<sup>4</sup>The cURL command line tool can be found on <https://curl.haxx.se/>

## Chapter 3

# Technologies and Frameworks

In this chapter will be discussed the details of technologies and frameworks that will be used developing the application. At first the Swagger framework will be introduced. In next sections the “front-end” web application framework Angular along with the superset of EcmaScript 6 – the TypeScript language will be presented. Following the thesis the web framework PatternFly will be introduced and its key features for designing web sites and web applications will be shown.

### 3.1 Introduction to Swagger Framework

Swagger is an open source framework backed by a large ecosystem of tools that helps developers design, build, document and test RESTful web services. It allows to describe the structure of the APIs so that machines can read them. By reading the API’s structure, the framework can automatically build an API documentation. It does this by asking the API to return a YAML or JSON that contains a detailed description of the entire API. The file is essentially a resource listing of the API which adheres to OpenAPI Specification<sup>1</sup>. Developers can write the specification for the API manually or have it generated automatically from annotations in the source code.

#### 3.1.1 Generating the API Documentation

To clarify how the Swagger works, let us consider an example. In the example can be seen use of Swagger on top of the Spring Framework. However keep in mind that the Swagger is a specification, and supports a wide range of frameworks. Swagger supports various annotation like the `@Api` for instance, that allows developers to specify details of the documentation.

```
1 @RestController
2 @RequestMapping("/users")
3 @Api(value = "users", description = "Users endpoints")
4 public class UsersController { ... }
```

Listing 3.1: An example of Swagger’s internal annotation that can be used to generate documentation.

---

<sup>1</sup>The OpenAPI Specification is a specification for “machine-readable” interface files for describing, producing, consuming, and visualizing RESTful web services.

The alternative way is to let Swagger figure out the documentation by itself based on the annotations from Spring Framework. Under the hood, it scans the Spring controllers on “start-up” and registers a documentation controller that exposes the operations the controller allows. 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.

```
1 @RestController
2 @RequestMapping("/users")
3 public class UsersController {
4
5     @Autowired
6     private UserService userService;
7
8     @PostMapping
9     @ResponseStatus(HttpStatus.CREATED)
10    public User createUser(@RequestBody UserDto userDto) {
11        return userService.createUser(userDto);
12    }
13
14    @GetMapping
15    @ResponseStatus(HttpStatus.OK)
16    public List<User> findAllUsers() {
17        return userService.findAllService();
18    }
19
20    @DeleteMapping
21    @ResponseStatus(HttpStatus.NO_CONTENT)
22    @RequestMapping(value = "/users/{userId}")
23    public void deleteUserById(@PathVariable Long userId) {
24        userService.deleteUserById(userId);
25    }
26
27 }
```

Listing 3.2: An example of Spring MVC controller that can be used by Swagger to generate API documentation.

The documentation then consists of 2 parts, as shown in the listing 3.3, the operations and the models. As can be seen a client can send a GET request on the `/users` endpoint to select the users. This is an example of an operation. Following the code, it can be seen that the operation returns a list of users of which a client can learn more in the models section.

However, the main advantage of the framework is that it exposes the endpoints from the documentation controller and makes them accessible on the `/api-docs` URL. Therefore the application can be built upon it, plus any developer who uses the framework, will be able to use the result of the thesis for testing his own application.

```

1 {
2   "apiVersion": "1.0",
3   "swaggerVersion": "1.0",
4   "basePath": "http://localhost:8080",
5   "resourcePath": "/users",
6   "apis": [
7     {
8       "path": "/users",
9       "description": "users",
10      "operations": [
11        {
12          "httpMethod": "GET",
13          "summary": "findAllUsers",
14          "notes": "",
15          "deprecated": "false",
16          "responseClass": "List [ User ]",
17        },
18      ],
19      ...
20    ]
21  },
22 ],
23 "models": [
24   "User": {
25     "properties": {
26       "id": {
27         "type": "long"
28       },
29       "firstname": {
30         "type": "string"
31       },
32       "lastname": {
33         "type": "string"
34       }
35     }
36   },
37   ...
38 ]
39 }
40

```

Listing 3.3: An example of documentation in JSON format generated by the Swagger framework from code in the listing 3.2.

## 3.2 What is Angular?

Angular [3] is an open source TypeScript framework used to build web applications in HTML and TypeScript. It makes it easy to build application as it combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges.

### 3.2.1 Beginning as AngularJS

Angular originally started as AngularJS when it was developed in 2009 by M. Hevery [4] 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 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<sup>2</sup>. However, because some of the core assumptions in AngularJS needed 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. **Components** – The earlier version of Angular had a focus of Controllers, but now has changed the focus to having components over controllers. They help to build applications into many modules, which helps in better maintaining the application over a period of time.
2. **TypeScript** – The newer version of Angular is based on TypeScript. It is a superset of JavaScript maintained by Microsoft. We will learn more about TypeScript later in section 3.3.
3. **Services** – are a set of code that can be shared by different components of an application. For example if we had a data component that picked data from a database, we could have it as a shared service that could be used across multiple applications.

### 3.2.2 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.

1. **Template** that is used to render the view for the applications. It contains the HTML that needs to be rendered as well as the bindings and directives.
2. **Class** is like a class defined in any language such as C except it is defined in TypeScript. Classes contains properties, methods, and the code which is used to support the view.
3. **Metadata** contains an extra data defined for the Angular class. They are defined with a decorator.

---

<sup>2</sup>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.

```

1 @Component ({
2   selector: 'my-app-hello-world',
3   template: `
4     <div>
5       <h1>{{ title }}</h1>
6     </div>
7   `,
8 })
9 export class HelloWorldComponent {
10   title: string = 'Hello World!';
11 }

```

Listing 3.4: An Angular class with the *@Component* decorator and a template.

In the listing 3.4 can be seen an example of the completed code with class, template and metadata. It defines a class in TypeScript called *HelloWorldComponent* that contains only one property – *title*. The component was defined using the *@Component* decorator that contains the HTML template, which is the view that needs to be rendered in the application.

The second cornerstone of an Angular application is dependency injection. The idea behind it is pretty simple. 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. By doing so, it is possible to depend on interfaces rather than concrete types. This approach leads to more decoupled code, which enables testability and other great things.

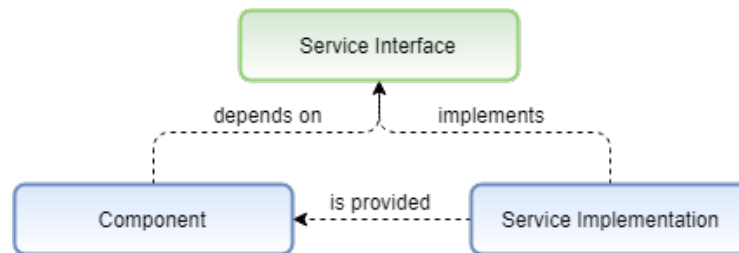


Figure 3.1: Angular’s way of injecting and instantiating services in the components.

To clarify, let us consider an example. In the listing 3.5 can be seen 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 specified in that component, Angular will automatically instantiate and inject the *UserService* 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<sup>3</sup> for example.

<sup>3</sup>The Spring Framework is an application framework and inversion of control container for the Java platform.

```

1 export class UserService {
2     users: User[] = [];
3
4     findUsers(): User[] {
5         // Here should be the code used to retrieve the users
6         return users;
7     }
8 }
9
10
11
12 @Component {
13     ...
14 }
15 export class UsersComponent {
16     users: User[] = [];
17
18     constructor(userService: UserService) {
19         this.users = userService.findUsers();
20     }
21 }

```

Listing 3.5: An example of simple dependency injection in Angular.

### 3.3 Introducing TypeScript

In September 1995 was first introduced JavaScript 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 the code. 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.

#### 3.3.1 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 safety 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 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 variable will be inferred by the TypeScript Language Service. All other “built-in” types and “user-defined” types inherit from the *any* type.

```

1 var foo = 42;
2 foo = 'Hello World!';
3
4 // The previous line will throw an error:
5 // cannot assign 'string' to 'number'

```

Listing 3.6: An example of TypeScript’s type inference. The type of the variable is inferred by the TypeScript Language Service based on its value.

Important to mention is 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.

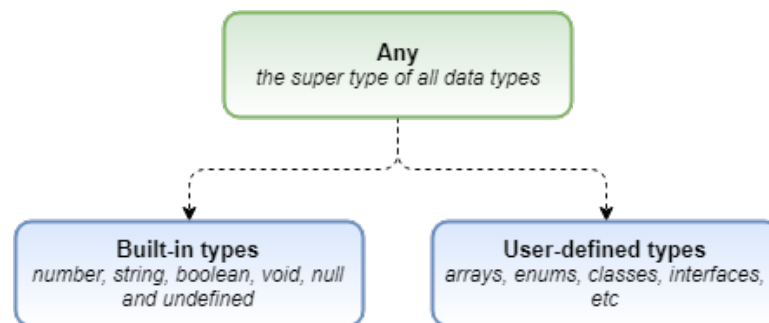


Figure 3.2: Data type classification in TypeScript.

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.

### 3.3.2 Future JavaScript

The second goal of TypeScript was to provide planned features from future JavaScript editions. Nowadays, it provides a number of features that are planned in EcmaScript 6<sup>4</sup> 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 a part of the specification.

In conclusion, even though 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.

<sup>4</sup>The EcmaScript specification is a standardized specification of a scripting language.

## 3.4 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 design is not done well, the whole application can be perceived as bad. The PatternFly framework was developed specifically to address this issue.

One of the main things that sets the framework apart from other libraries 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 to unify disparate projects, make them look great and make them look like they belong in the same portfolio.

PatternFly is an open source project that is based on Bootstrap [7], 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 many other techniques which allow developers create code that is more maintainable, themeable and extendible. This allows the developer 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 implement various components of the toolkit. These stylesheets are generally compiled into a bundle and included in web pages, but individual components can be included or removed. Moreover it provides a number of variables that control things such as color and padding of various components. Each component consists of a HTML structure and CSS declarations, and in some cases accompanying JavaScript code.

### 3.4.1 Using the Components

PatternFly comes with various design templates for typography, tables, forms, buttons, navigation, and other interface components that can be used building the application – saving lots of time and efforts in the process. These 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 names like *.alert* and *.alert-success*, these components are easily reusable and extensible. Although the framework uses descriptive class names that have meaning, it is not specific about implementation details. Therefore all classes can be overridden with custom CSS style and still the meaning of the class will remain the same.

```
1 <div class="container">
2   <div class="alert alert-success">
3     <span class="pficon pficon-ok"></span>
4     <strong>Hello World!</strong>
5     <span>This is an example of an alert in PatternFly</span>
6   </div>
7 </div>
```

Listing 3.7: An example of *alert* implemented using the PatternFly framework.

To clarify, the following code snippet in the listing 3.7 will generate an *alert* component with 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 3.3: The result of styling the components with PatternFly using the code in listing 3.7

### 3.4.2 Working with the Grid

In the beginning 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. This 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. Grids provide structure to the layout, defining the horizontal and vertical guidelines for arranging content and enforcing margins.

To use the grid system, a few rules need to be followed. Grid column elements are placed inside row elements, which create horizontal groups of columns. It is possible to have as many rows as we want on the page, but columns must be immediate children of rows. In a full row, the column widths will be any combination that adds up to 12, but it is not mandatory to use all available columns.

```
1 <div class="container">
2   <div class="row">
3     <div class="col-md-6">First column</div>
4     <div class="col-md-6">Second column</div>
5   </div>
6   <div class="row">
7     <div class="col-md-3">First column</div>
8     <div class="col-md-3">Second column</div>
9     <div class="col-md-3">Third column</div>
10    <div class="col-md-3">Fourth column</div>
11  </div>
12 </div>
```

Listing 3.8: An example of the grid system in PatternFly.

The rows needs to be places in a “fixed-width” layout wrapper that has a *.container* class and a width of 1170px or in “full-width” layout wrapper, which has a *.container-fluid* class, and which enables the responsive behavior in that row. The grid system is based on four tiers of classes – *xs* for phones, *sm* for tablets, *md* for desktops and *lg* for larger desktops. These classes define the sizes at which the columns will collapse or spread horizontally. To clarify, let us consider an example with two rows, one with two columns, and one with four, such as in listing 3.8.

## Chapter 4

# Application Design

This chapter will explain the application requirements and cover the application design drafts in detail. The drafts consists of separate UI elements that, once combined, will represent the final design draft of the application. The complete design drafts can be found in the Chapter [A](#).

### 4.1 Application's Requirements

As mentioned in the beginning, the aim of the thesis is to design an application that will allow to create test cases from an application's API. Therefore it has to allow the user to choose the application's API upon which the application developed in the thesis will work. When the user chooses the API, the application should be able to present the API's endpoints to the user and allow their testing. Once it has access to the API, it is needed to specify the most important requirement of the application – creating the test cases.

It should allow creating test cases that can combine various endpoints and other test cases. To clarify, it should be possible to test an API's endpoint and use its output as an input of another endpoint. Last but not least, once the users create some test cases, they should be able to view and run them. Therefore the application should be able to present, filter, sort, and run them.

### 4.2 Designing the Project Explorer

This section focuses on describing the design process of the Project Explorer. The Project Explorer will allow users to create, manage and remove projects in the application. As can be seen in the figure [4.1](#) the project consists of it's name and a source of the API. As the source of the API is considered an URL (or a link to the local file system) of a Swagger's JSON file that contains the API's documentation.

When the project is added to the application via the Project Explorer's modal window, it should appear in the Explorer's table as can be seen in the figure [A.3](#). Table consists of an enumeration of the projects and their attributes, e.g. amount of endpoints or test cases that in the project.

The Project Explorer view will allow users to manage various projects, however, to keep things simple the users will not need to view the Explorer for switching between projects. As can be seen in the figure [A.4](#), the application will contain the navigation bar that will allow users swiftly switch between projects without accessing the Project Explorer.

Figure 4.1: A modal window for adding projects to the application.

### 4.3 Showing the Endpoints

When the users are able to add a project to the application. They should be able to view and test API endpoints that are provided by that project. Many applications that provide overview of an API endpoints does not scale well enough if they encounter large amount of the endpoints. To address this issue the “tree-like” structure was used for the presentation of the API.

Figure 4.2: A leaf node of a tree structure that is used for displaying the information about API endpoints.



As can be seen in figure A.5 by default only “top-level” nodes of the tree are shown. The users can then expand the nodes and view details of the particular endpoints or run tests using the settings menu in the top right corner of the endpoint node. When the user encounters the leaf node, the information about that particular leaf node are shown in detail as can be seen in the figure 4.2.

## 4.4 Creating the Test Case

Even though it is important to present the API endpoints to the users in a friendly way, the main focus of the application lies in the test cases. Therefore the Test Case Creator was designed. The design of the Creator is based on canvas, which is a HTML5 element that allows for dynamic, scriptable rendering of 2D shapes and bitmap images. The Creator is designed as an dynamic editor that allows users to build test cases using various endpoints, test cases and control structures. The idea behind it is that the users will be able to construct complex test cases in the same way as they would create state machine diagrams<sup>1</sup>.

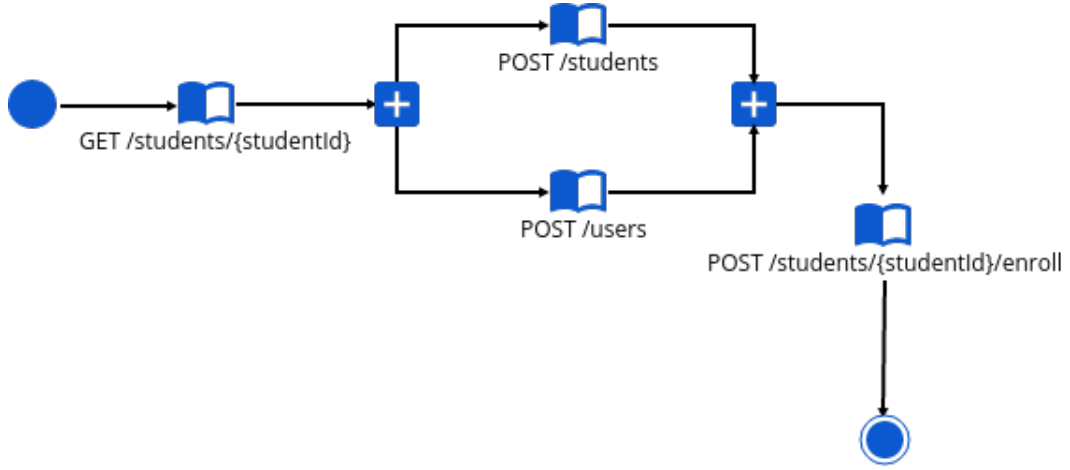


Figure 4.3: An example of a test case in the format of a state machine diagram created using the Test Case Creator

As can be seen in the figure A.6 the editor consists of several parts. All of them together allows users to model desired test cases using the drag and drop functionality. To clarify, consider an example shown in the figure 4.3. In the example, using the drag and drop functionality was inserted several endpoints to the canvas. When an endpoint is inserted, a modal window with endpoint’s details appears. Either the endpoint details can be filled using the form in the window or using another endpoint’s output, simply connecting them in the canvas. The figure is modelling simple test case that will find a student by given id, uses its information details as an input for creating new student and a user, and then enrolling the newly created student to some course.

When the test cases are made using the Test Case Creator, the users should be allow to view, manage, edit, and run them. Hence the test cases overview was designed. As can be seen in the figure A.7 overview consists of a navigation bar with buttons that allow

<sup>1</sup>A state machine diagram is a behavior diagram which shows discrete behavior of a part of designed system through finite state transitions.

managing the test cases, e.g. creating, running or removing them. Following the design, the table with test cases was created. This allows users to view, sort and filter all test cases in the project. The name of the test case in the table is then used to create a link that enables users to edit the test case in the Test Case Creator.

## Chapter 5

# Conclusion

The aim of this term project was to design an application that will allow its users to create test cases from other applications API. As a part of working on the above goal the technologies and frameworks that will be needed when building the application upon the design were presented.

On top of that using patterns from the PatternFly framework the first application's designs were created. The most important part of the application – the Test Case Creator was designed. An interactive editor that uses simplified state machine diagrams to allow users to build the test cases easily and fastly.

For the future, the following improvements are planned such as redesign of the endpoint's details that would allow the users to fill in some default data for that particular endpoint, which would allow the users to add endpoints directly to the test cases without the need to specify the endpoint's data.

Another key step is to implement the application using the technologies listed in Chapter 3 and to test it using the Red Hat JBoss BPM Suite as a test project.

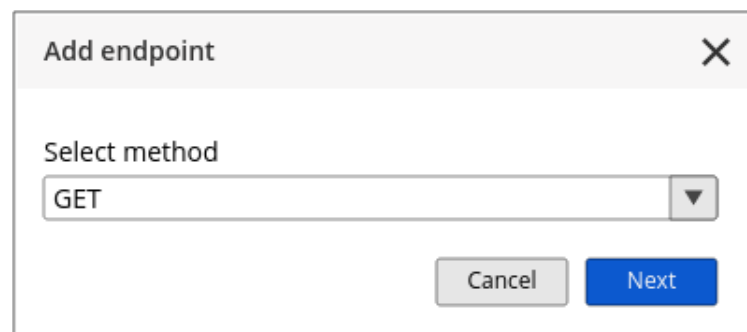
# Bibliography

- [1] Fielding, R. T.: Architectural Styles and the Design of Network-based Software Architectures [dissertation]. University of California, Irvine. 2000.
- [2] 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>
- [3] Frisbie, M.: *Angular 2 Cookbook*. Packt Publishing. 2017. ISBN 978-1785881923.
- [4] 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/>
- [5] Leonard Richardson, S. R., Mike Amundsen: *RESTful Web APIs: Services for a Changing World*. O'Reilly Media. 2013. ISBN 978-1449358068.
- [6] SmartBear Software: Swagger, the world's most popular API tooling. [Online; visited 10.12.2017].  
Retrieved from: <https://swagger.io/>
- [7] Spurlock, J.: *Bootstrap: Responsive Web Development*. O'Reilly Media. 2013. ISBN 978-1449343910.
- [8] 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

# The Design Drafts

This chapter contains complete design drafts of the application.



The image shows a software dialog box titled "Add endpoint". It features a close button (an 'X' icon) in the top right corner. Below the title bar, the text "Select method" is displayed above a dropdown menu. The dropdown menu is open, showing the selected option "GET" and a small downward-pointing arrow on its right side. At the bottom right of the dialog, there are two buttons: a grey "Cancel" button and a blue "Next" button.

Figure A.1: Adding endpoint to a test case – selecting the endpoint’s HTTP method.

Add endpoint

×

Headers +

Header

Value

Header

Value

\* Path Variables

Variable

Variable

Attributes

Attribute

Attribute

Query Parameters

Parameter

Parameter

Back

Cancel

Save

Figure A.2: Adding endpoint to a test case – form for filling the endpoint’s data.

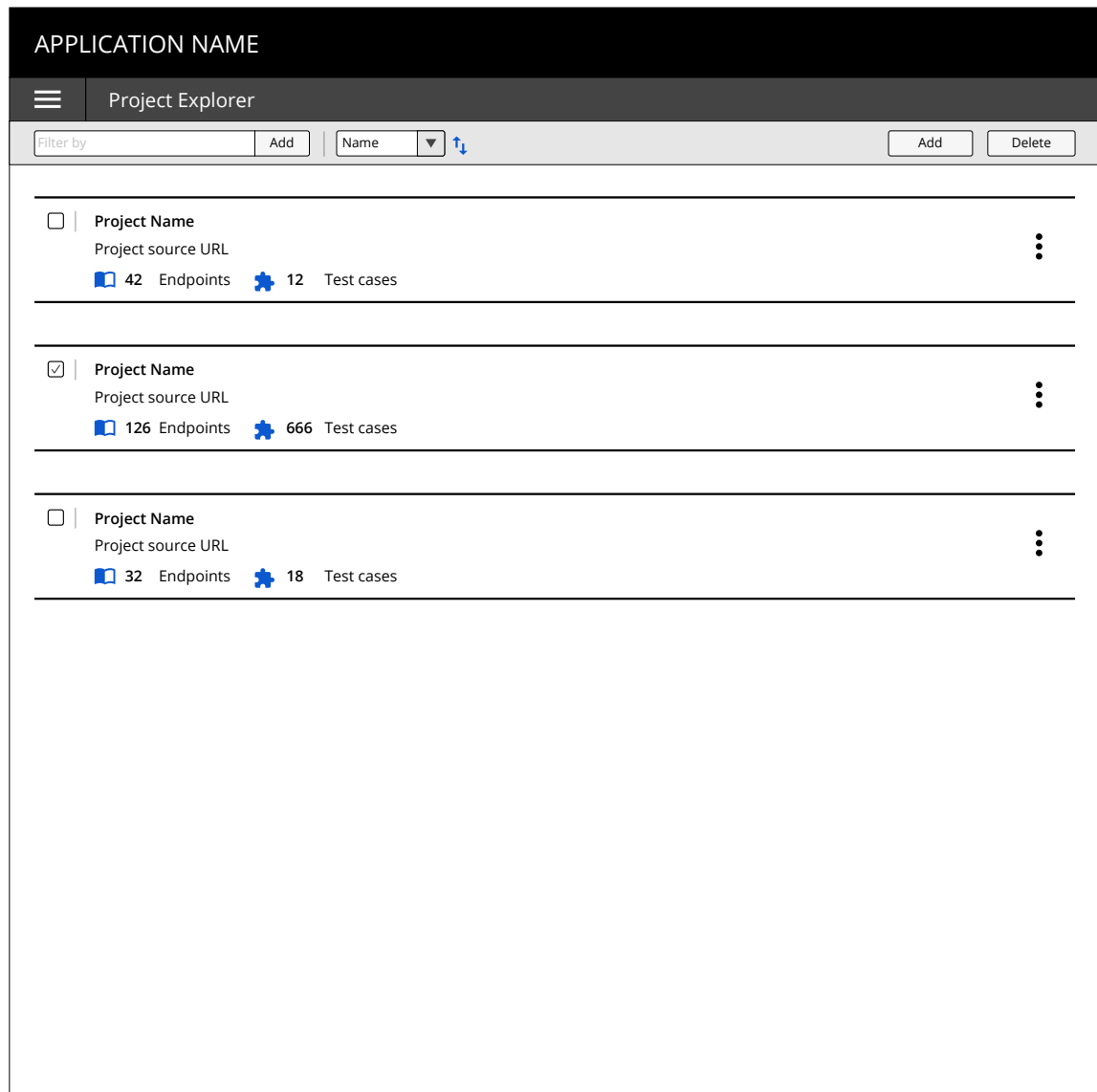


Figure A.3: The Project Explorer page which contains information about projects added to the application.

APPLICATION NAME

project-name

+ Add test case

Overview

API

Test cases

List by Most recent

TEST CASE

Name of the test case

DESCRIPTION

Description of the test case

Last test run: 1. 1. 2018 ✓

TEST CASE

Name of the test case

DESCRIPTION

Description of the test case

Last test run: 1. 1. 2018 ✓

ENDPOINTS

API

URL of the API

Last test run: 1. 1. 2018 ✗

ENDPOINT

API

HttpMethod URL of the API

Last test run: 1. 1. 2018 ✗

Figure A.4: The Dashboard of the application that contains recently tested endpoints and test cases.



APPLICATION NAME

project-name

+ Add test case

Overview

API

Test cases

Filter by

Add

ENDPOINTS

>

API /users

Last test run: 1. 1. 2018

✓

ENDPOINTS

✓

API /students

Last test run: 1. 1. 2018

✗

>

API GET /students

Last test run: 1. 1. 2018

✗

>

API POST /students

Last test run: 1. 1. 2018

✓

ENDPOINT

✓

API POST /subjects

Last test run: 1. 1. 2018

✗

REQUEST

Headers

> Content-Type: application/json

Body

> subjectName: string

> minStudents: number

RESPONSE

Status 200

Headers

> Content-Type: application/json

Body

> id: number

> subjectName: string

> minStudents: number

Figure A.5: The page with API endpoints ordered into a hierarchical structure.

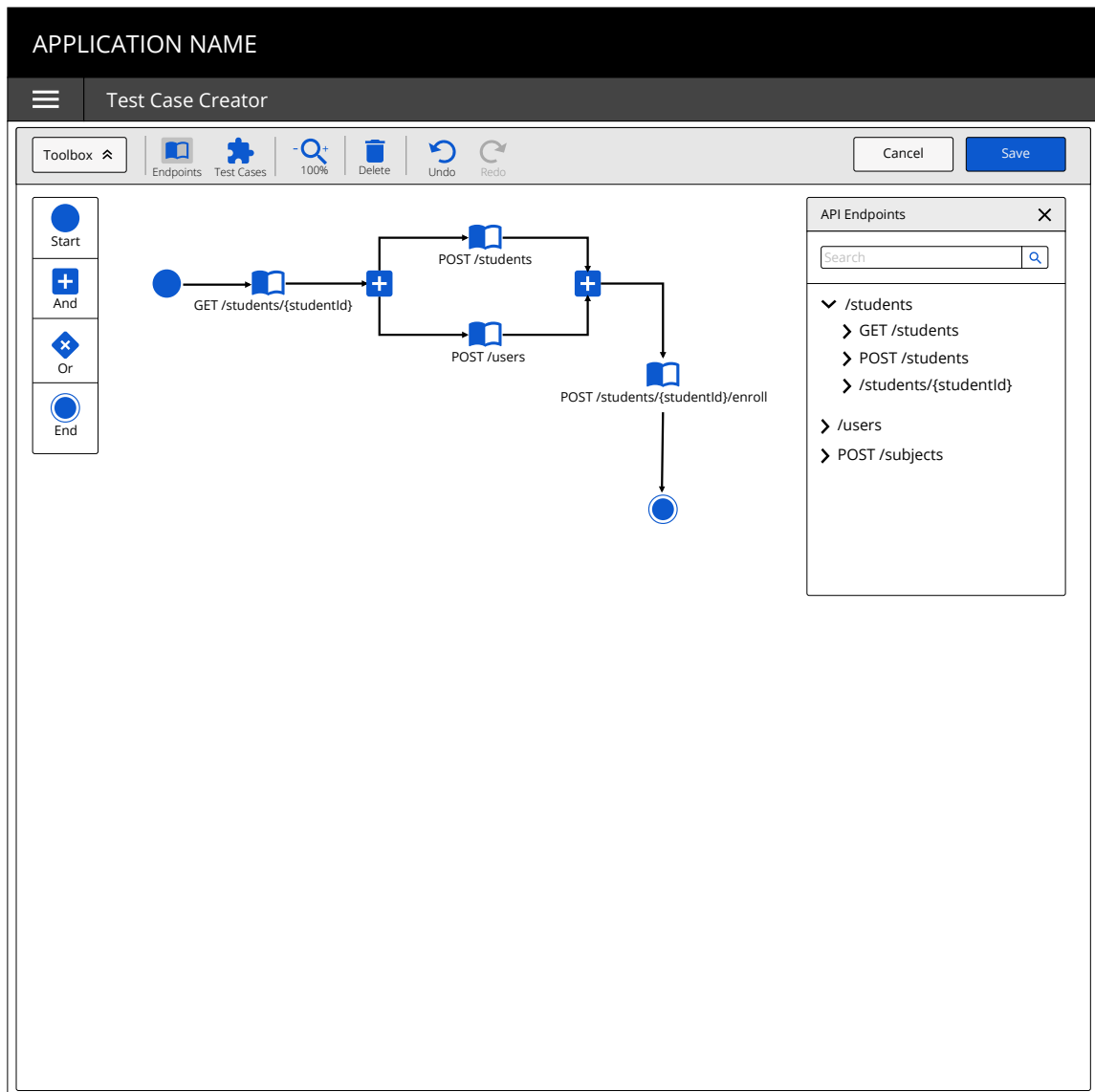


Figure A.6: The design of the Test Case Creator that allows creating test cases from application's API and other test cases.

APPLICATION NAME

project-name

+ Add test case

> Overview

> API

> Test cases

Filter by

Add

Add

Run

Delete

5 Results

Active Filters:

Name: test case

Run after: 31. 12. 2017

Clear all filters

| Name   | Description                      | Last run    | Status |
|--|----------------------------------|-------------|--------|
| <input type="checkbox"/> Test case number 1    | This is the first test case...   | 1. 1. 2018  | ✗      |
| <input type="checkbox"/> Test case number 2    |                                  | 12. 1. 2018 | ✓      |
| <input type="checkbox"/> Test case of students |                                  | 3. 1. 2018  | ✗      |
| <input type="checkbox"/> Test case of users    | Test case of users endpoints.    | 14. 2. 2018 | ✓      |
| <input type="checkbox"/> Subject's test case   | Test case for subjects endpoint. | 15. 3. 2018 | ✓      |

Select all

15

per page

1 - 5 of 5

<<

1

of 1

>>

Figure A.7: The design of the test cases overview that contains information about created test cases inside a table.