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 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, Test automation

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. Master's thesis. 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
March 12, 2018

Acknowledgements

I would like to thank Mr. Mgr. 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

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 the products and create an ecosystem of shared knowledge that allows other developers to use the code provided by the APIs. To fulfill these tasks, APIs 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.

This thesis aims to solve the problem of documentating 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 an API reference, but mainly the functionality to create extensive test cases from the listed web services.

The thesis is organizes 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 designs, wireframes and mockups which can help reveal any clashing visual elements before writing the code. The last Chapter contains an overall summary of the designed 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 is introduced basic terminology and established notion of remote interfaces and web services. In the next section is provided an explanation of what APIs are and the last section covers the importance of theirs 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 [3], allowing less costly interactions over the Internet. However, in 2004 the W3C extended the definition of web services about "REST-complient" web services [11], 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 [2]. 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, while JSON being currently the most popular format being used.

2.1.2 Messaging

As mentioned in 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.

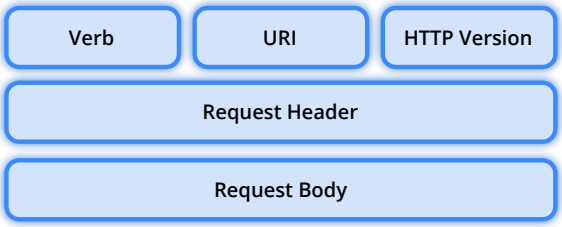

Figure 2.1: The format of a HTTP Request.

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.

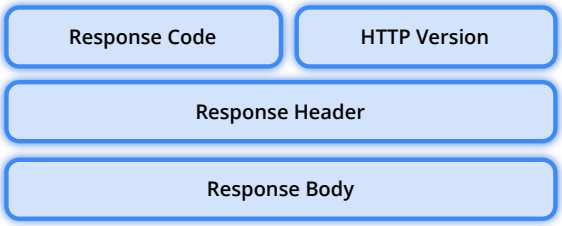

Figure 2.2: The format of HTTP response.

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 [6] 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 [7]. It is being more and more 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 the 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. On 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 to test 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 not a surprise that many frameworks and applications for theirs testing were developed. Probably the oldest of them is cURL [10]. 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 will be discussed the details of technologies and frameworks that will be used during the development of the application. At first the Swagger framework will be introduced. In the 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 and its key features will be introduced.

3.1 Introduction to Swagger Framework

Swagger [8] 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.

3.1.1 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 3.1, 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[1]. Under the hood, it scans Spring

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

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 3.1: 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”. As can be seen in the listing 3.2, 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 3.2: An example of API documentation in JSON format created using the Swagger framework.

3.2 What is Angular?

Angular [4] 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.

3.2.1 Beginning as AngularJS

Angular originally started as AngularJS, it was developed in 2009 by M. Hevery [5] 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[2]. 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. 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 3.3.

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.

3.2.2 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 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 3.3 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.

[2] 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.

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

Listing 3.3: 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. 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.

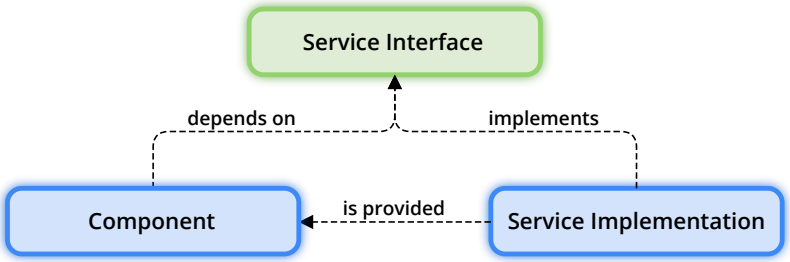

Figure 3.1: Angular's way of injecting and instantiating services in the components.

To clarify, consider an example from the listing 3.4. 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[3].

[3]Spring framework is an application framework and inversion of control container for the Java platform that relies heavily on dependency injection.

```
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 3.4: An example of 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. 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 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 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.

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.

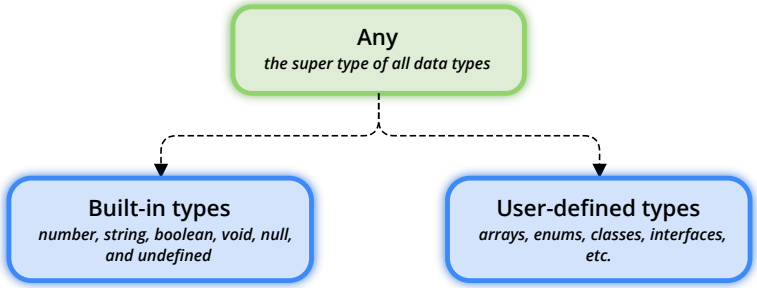

Figure 3.2: Classification of data types in TypeScript.

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.

3.3.2 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[4] 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.

3.4 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's design is not done well, whole application can be perceived as bad. The PatternFly framework was developed specifically to address the 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 to unify disparate projects, make them look great and make them look like they belong in the same portfolio.

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

PatternFly is an open source project that is based on Bootstrap [9], 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.

3.4.1 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 3.5: An example of styling the component using predefined class *.alert*.

As can be seen in the figure 3.3, the code snippet from the listing 3.5 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 3.3: Illustration of a component styled using the semantic classes from PatternFly.

3.4.2 Working with the Grid

In the beginning of this section 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.

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 3.6: An illustration of the grid system in PatternFly.

The rows have to be places 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.

Chapter 4

Application Design

This chapter will go over the application requirements and will cover the application design drafts in detail. The design drafts 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 drafts consists of separate UI elements that, once combined, will represent the final design draft of the application.

4.1 Application's Requirements

As mentioned in the beginning, the aim of the thesis is to degign an application that will allow to create test cases from an other application's API. The application's API will be provided using the Swagger's documentation file, which will contain information about the API. Therefore the application developed in the thesis has to allow the user to choose the application's API upon which it will work. When the user chooses the file with the API, the application should be able to present the API's endpoints to the user and allow their testing.

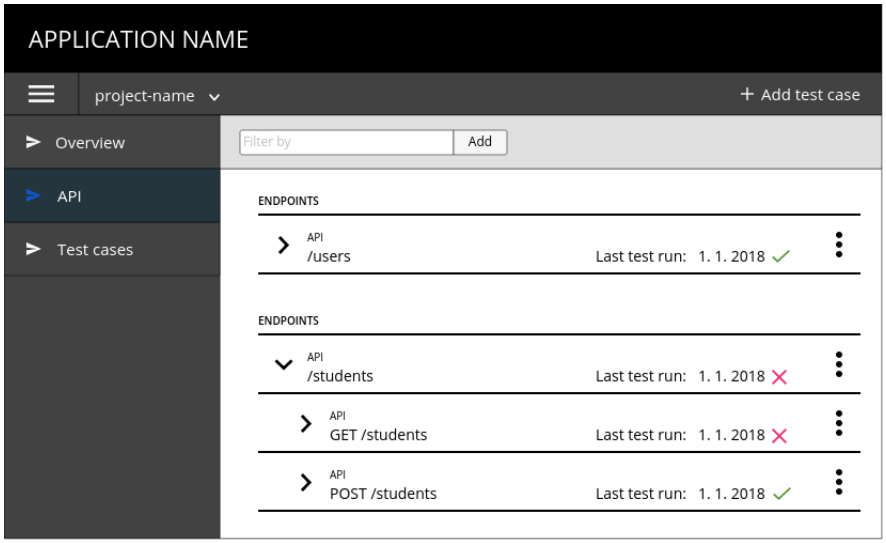

Figure 4.1: The Dashboard of the application that contains the navigation bar, menu and an overview of recently run endpoints and test cases.

Last but not least, it is needed to specify the most important requirement for the application – creating the test cases. The application should allow creating complex test cases that can combine various endpoints and other test cases. To clarift, it should be possible to test an API's endpoint and use its output as an input of another endpoint. Moreover, once the users create some test cases, they should be able to manage them. Therfore 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.

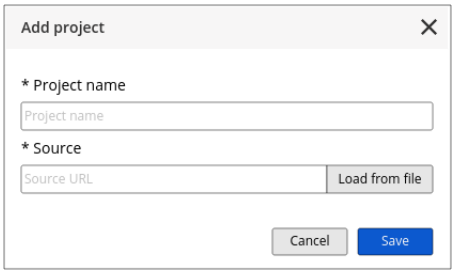

Figure 4.2: A modal window for adding projects to the application.

As can be seen in the figure 4.2 each 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 Project Explorer's table as can be seen in the figure 4.3. The table consists of an enumeration of the projects and their attributes, e.g. amount of endpoints or test cases in the project.

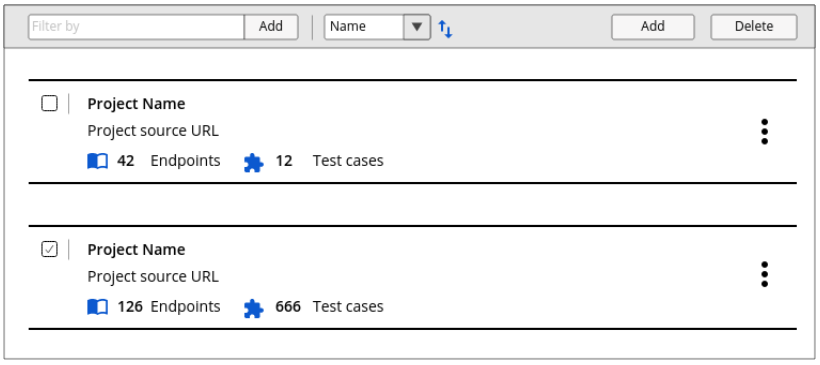

Figure 4.3: The Project Explorer that contains information about projects that were added to the application.

The table view will allows users to manage various projects, however, to keep things simple the users will not need to view the Project Explorer for switching between projects. As can be seen in the figure 4.1, which shows the application's dashboard, the application will contain the navigation bar that will allow users to swiftly switch between projects without accessing the Project Explorer.

4.3 Showing the Endpoints

Once the users choose a project to work with within the application, they should be able to view and test the endpoints that are provided by that project. However, as explained in the section 2.3, many applications which provide an overview of an API's endpoints does not scale well enough if they encounter large amount of the endpoints. To address this issue, in the thesis was used for the presentation of the APIs a "tree-like" structure.

Figure 4.4: The overview of an API's endpoints ordered into a hierarchical "tree-like" structure.

As can be seen in figure 4.4 by default only "top-level" nodes of the tree are shown. However, the users can then expand the nodes and view details of particular endpoints or run tests using the settings menu in the top right corner of the endpoint node. Once the user encounters the leaf node, the information about that particural leaf node are shown in detail as can be seen in the figure 4.5.

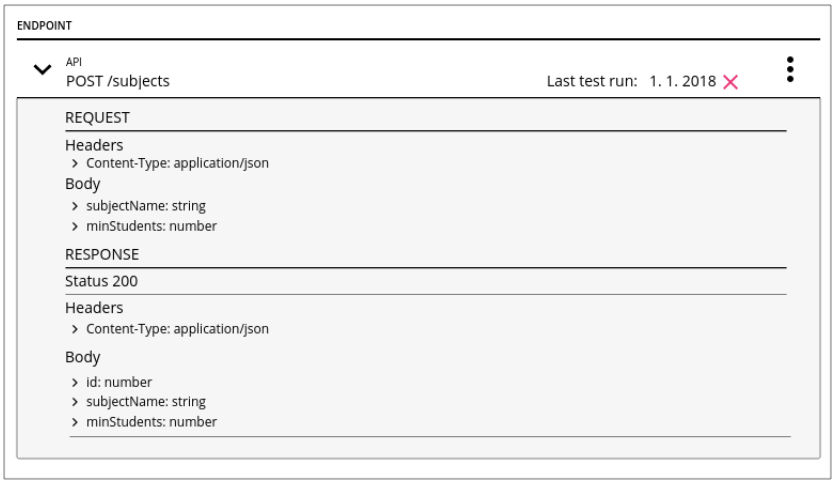

Figure 4.5: A leaf node of a hierarchical structure that is used for displaying the information about API's endpoints.

4.4 Creating the Test Case

Even though it is important to present the API's endpoints to the users in a friendly way, the main focus of the application lies in the test cases. For that particular reason the Test Case Creator was designed. The design is based on canvas, which is a HTML 5 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[1].

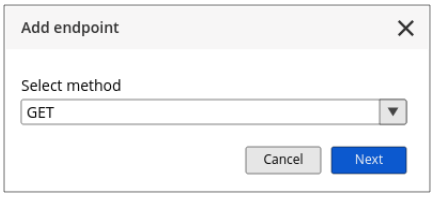

Figure 4.6: A modal window for adding endpoint to a test case – selecting the endpoint's HTTP method.

The Creator 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 4.7. In the example, using the drag and drop functionality was inserted several endpoints to the canvas. When an endpoint is dropped to the canvas a modal window with endpoint's settings appears. If the dropped endpoint was not a leaf node the modal window asks the user to select endpoint's method as shown in the figure 4.6.

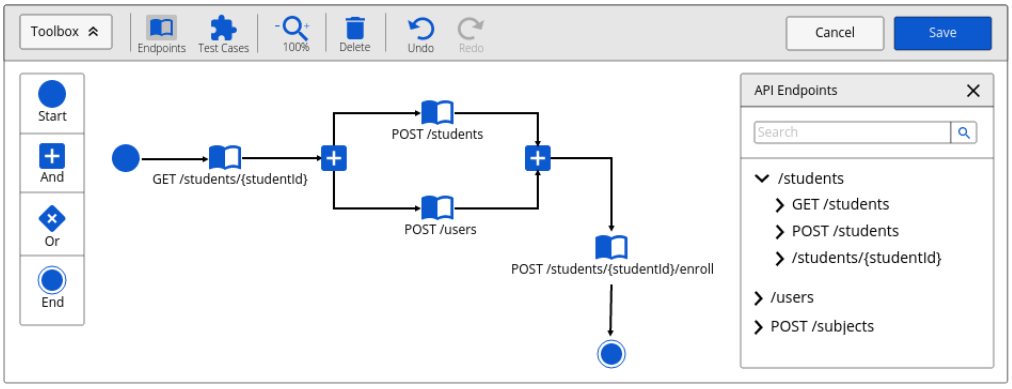

Figure 4.7: The Test Case Creator with an example of a test case in the format of as state machine diagram.

However, if the endpoint was a leaf node the modal window with endpoint's detail is shown as can be seen in the figure 4.8. Then either the endpoint's details can be filled using the form in the modal or skipping the form and using another endpoint's output as the endpoint's details – simply connecting them in the canvas. Overall, the figure 4.7 is modelling simple test case that will find a student by given id, uses its information details

[1] A state machine diagram is a behavior diagram which showsh discrete behavior of a part of designed system through finite state transitions.

as an input for creating new student and a user, and tehn enrolling the newly created student to some course.

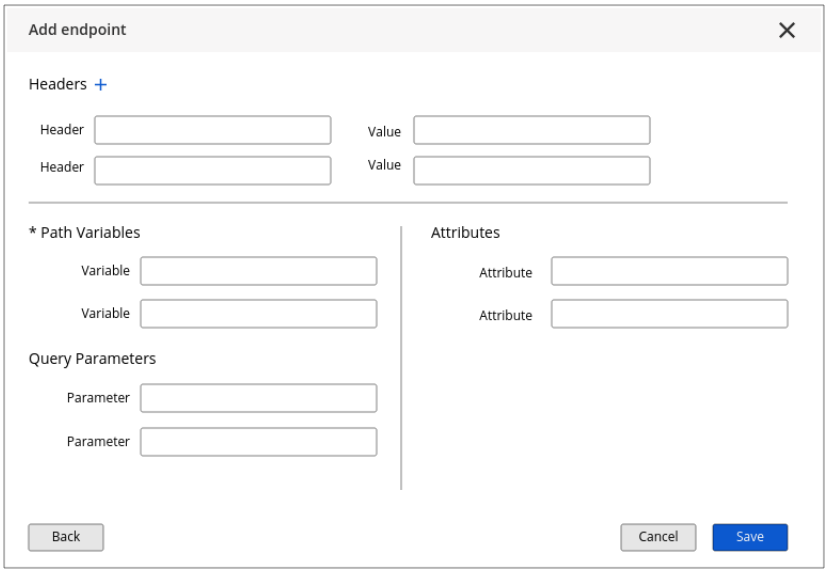

Figure 4.8: A modal window for adding endpoint to a test case – filling the endpoint's information.

Once the test case is creates using the Creator, the users should be allow to view, edit, and run it. Hence the Test Cases Overview was designed. As can be seen in the figure 4.9, the overview consists of a navigation bar with buttons that allow managing the test cases, e.g. creating, running or removing them. Following the design, the table with test cases was created. The table allows users to view, sort and filter all test cases in the project. Each test case in the table consists of basic information about it, such as the day when it was last run or whether the last run was successful or not. 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 Creator.

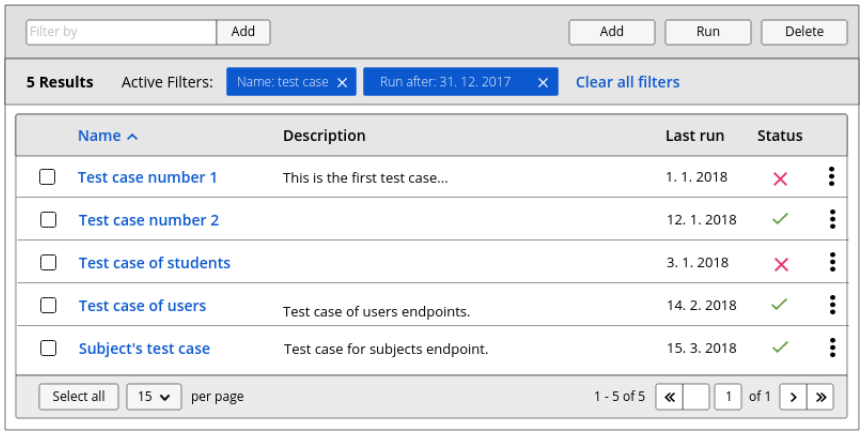

Figure 4.9: The Test Cases Overview that contains information about created test cases.

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 and the most important part of the application – the Test Case Creator was designed as 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] 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] Fielding, R. T.: Architectural Styles and the Design of Network-based Software Architectures [dissertation]. University of California, Irvine. 2000.

[3] 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

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

[5] 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/

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

[7] 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.

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

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

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

[11] 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