Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

MIGRATION OF THE USER INTERFACE OF A WEB APPLICATION

- From Thymeleaf to Angular

Jennie Eriksson

2022:02

Datum för godkännande: 09.03.2022
Handledare: Björn-Erik Zetterman

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Jennie Eriksson
Arbetets namn:	Migrering av användargränssnitt för en webbapplikation
Handledare:	Björn-Erik Zetterman
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Syftet med uppdraget är att införa ramverket Angular till Crosskey:s webbapplikation crosskey.io. Applikationens användargränssnitt ska på så sätt serveras på klientsidan med Angular istället för, som i nuläget, på serversidan med Thymeleaf.

Utvecklingen av användargränssnittet genomfördes med hjälp av programmeringsspråket TypeScript och ramverket Angular. Backend-applikationen utvecklades med hjälp av Java och ramverket Spring Boot. För implementering av REST API:t användes verktyget Swagger.

Resultatet av konverteringen till Angular blev ett användargränssnitt med högre prestanda och snabbhet än tidigare.

Nyckelord (sökord)

Angular, TypeScript, Thymeleaf, användargränssnitt, SPA, Selenium, Spring Boot, Java, REST, Swagger

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
2022:02	1458-1531	Engelska	40 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
08.02.2022	01.03.2022	09.03.2022

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Jennie Eriksson
Title:	Migration of the User Interface for a Web Application
Academic Supervisor:	Björn-Erik Zetterman
Commissioned by:	Crosskey Banking Solutions

Abstract

The purpose of this assignment is to introduce the Angular framework to Crosskey's web application crosskey.io. The user interface of the application will thus be served on the client side with Angular instead of, as at present, on the server side with Thymeleaf.

For the development of the user interface, the programming language TypeScript and the Angular framework was used. The backend application was developed using Java and the Spring Boot framework. The REST API was implemented with the help of Swagger tools.

The result of the conversion to Angular was a user interface with higher performance and speed than before.

Keywords

Angular, TypeScript, Thymeleaf, user interface, SPA, Selenium, Spring Boot, Java, REST, Swagger

Serial number:	ISSN:	Language:	Number of pages:
2022:02	1458-1531	English	40 pages

Handed in:	Date of presentation:	Approved:
08.02.2022	01.03.2022	09.03.2022

TABLE OF CONTENTS

1. INTRODUCTION

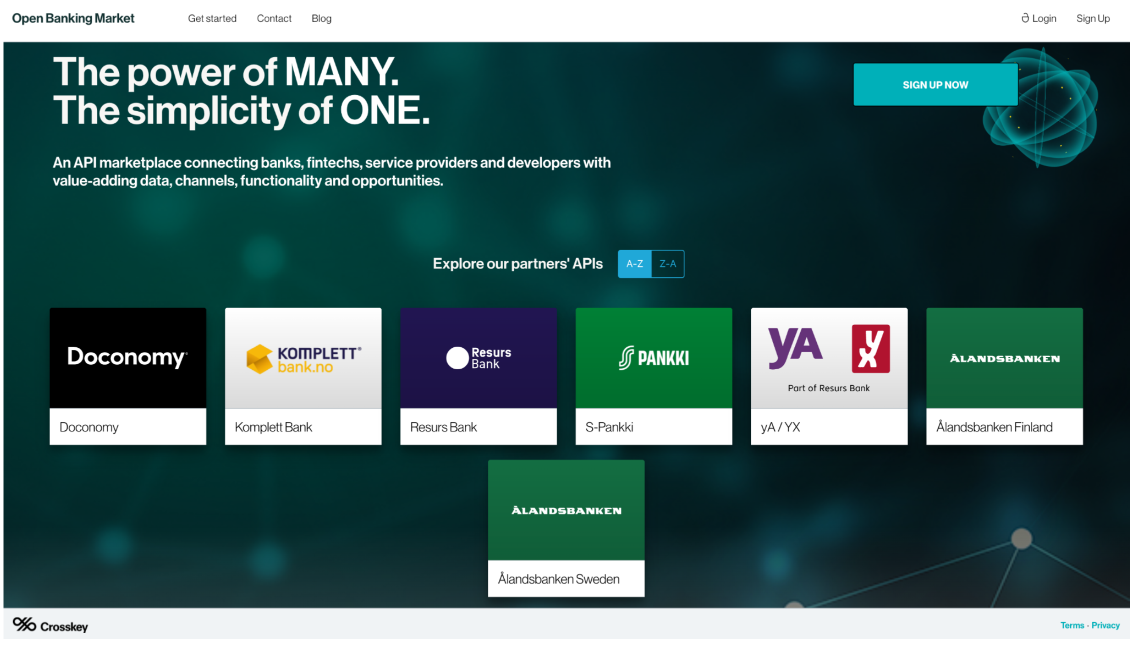
1.1 Background

1.1.1 Crosskey

The commissioner for this assignment is Crosskey Banking Solutions. Crosskey is an IT company and wholly owned subsidiary of Ålandsbanken Abp. Crosskey's area of business is delivering IT services to banks and other actors conducting banking, card or capital market operations. The company has over 300 employees who work at offices situated in Mariehamn, Stockholm, Turku and Helsinki (*About Crosskey*, 2015).

1.1.2 Crosskey.io

Crosskey's C°OPEN platform, Crosskey.io, is a cloud-based open platform for developers, banks and fintech companies, wishing to act as providers for different types of open banking services, such as account information and payment initiation services. The platform offers a market of open banking APIs that connects banks and third parties with data, channels and functionality. It provides a simple and cost effective way for banks and financial institutions to collaborate and integrate their products and services (*Open Banking - PSD2 as a Service*, 2018). Figure 1 is a screenshot of the homepage of the crosskey.io web platform.



The power of MANY.
The simplicity of ONE.

An API marketplace connecting banks, fintechs, service providers and developers with value-adding data, channels, functionality and opportunities.

Figure 1. The homepage of crosskey.io (Open Banking Market-Crosskey, n.d.).

1.2 Purpose

The purpose of this assignment is to migrate the user interface (UI) of Crosskey's[1] web application crosskey.io from server-side/backend to client-side/frontend. This involves a change in technology as well as the architecture of the system. The system will go from only consisting of a server application that serves the user interface using the template engine Thymeleaf, to serving the user interface as a separate client application running on the browser using the Angular framework.

The main reason for this shift towards a client-side rendered web application, is to increase the performance of the UI, thereby providing a better experience for the users of the website.

1.3 Requirement specification

Listed below are the pages that *must be* converted to Angular UI:

1. The homepage.
2. The tutorial page, which contains instructions on how to get started using an API.
3. The APIs page, which displays the APIs of a bank selected on the homepage.
4. The terms and conditions page.
5. The privacy page.
6. The dashboard page, which a user is directed to after successful login.
7. The teams page, which contains all the teams a user is a member of.
8. The team page, which is showing info about a specific team selected from the teams page.
9. The applications page, which contains all the applications a user has subscribed to.

Listed below are the pages that *would be nice to have* converted to Angular UI:

10. The contact page.
11. The register page.

[1] Crosskey Banking Solutions, https://www.crosskey.fi/

1.4 Methodology

The development work will be done in small increments or steps, where one page at a time is completed before another one is started. After an initial version of a page is complete, the code written for that page will be reviewed by members of my team, who will provide feedback and suggestions for improvements. Based on the feedback, I will then refine the code and again put it up for review. These steps will be repeated until a final version of the page is approved and moved to the test environment. In the test environment the process of feedback and refinement is renewed, since my team now can test the new version of the UI.

This cycle of feedback and refinement follows the method of Evolutionary Prototyping. The idea with this method is to improve the quality of the software by thorough testing. This way errors or missing functionality can be detected early in the process (Davis, 1992).

The code for the backend application will be written in Java using the Spring Boot framework. The frontend application will be written in Typescript using the Angular framework. Intellij will be used as the development environment since it is the standard IDE used at Crosskey.

1.5 Definitions

The list below gives a short description of some commonly used terms and abbreviations used in this essay:

- **API:** Application Programming Interface. An API is a set of definitions and protocols that explains how applications communicate with each other (*What Is a REST API?*, 2020).
- **Backend:** Represent the data access layer in a computer system responsible for storing and managing data (Techopedia, 2011).
- **Client:** A system that accesses remote services from a server (van Mulligen & Timmers, 1994).
- **CSR:** Client-Side Rendering.

- **CSS:** Cascading Style Sheets. Language used to describe the appearance of HTML documents (Krause, 2016).
- **DOM:** Document Object Model. Defines the logical structure and content of a web document, and how a document is accessed and manipulated (*Introduction to the DOM*, n.d.).
- **Frontend:** Represent the presentation layer in a computer system that interacts directly with the end user (Techopedia, 2011).
- **HTML:** Hyper Text Markup Language. The standard language for describing the structure of a web page (Krause, 2016).
- **IDE:** Integrated Development Environment.
- **JavaScript:** Language for defining the functionality/behavior of a web page (Krause, 2016).
- **JIRA:** Issue tracking tool used in agile development that helps teams plan and manage their work (Atlassian, n.d.).
- **JSON:** JavaScript Object Notation. Text format used for storing and transporting data objects (*JSON*, n.d.).
- **OAS:** Open API Specifikation.
- **Server:** A system that provides various functionality to other systems (van Mulligen & Timmers, 1994).
- **SASS:** Syntactically Awesome Style Sheets.
- **SEO:** Search Engine Optimization.
- **SPA:** Single Page Application.
- **SSR:** Server-Side Rendering.
- **UI:** User Interface.
- **YAML:** Yet Another Markup Language. Human readable data serialization standard (*The Official YAML Web Site*, n.d.).

2. ARCHITECTURES AND DESIGN PATTERNS

2.1 REST

REST stands for **RE**presentational **S**tate **T**ransfer and is an architecture style that defines a set of guidelines for creating web APIs. A web API is an interface that defines how web-based applications communicate with each other. APIs complying with the REST architecture are known as RESTful APIs or REST APIs. A RESTful system consists of a client which requests for the resources and a server, which provides the resources. RESTful systems interact by using HTTP (HyperText Transfer Protocol) (*What Is a REST API?*, 2020).

Two important constraints of RESTful systems are statelessness and that they implement a client-server architecture.

A system is stateless if the server and the client can communicate without knowing each other's state. This means that a request coming from the client contains in itself all the information the server needs to handle the request, which eliminates the need to store data from previous states in the server.

In a RESTful system the client and the server are implemented as separate applications. This means that changes done in the client code won't affect the server and vice versa.

These constraints allow the components in a RESTful system to be managed, modified and reused independently, thus increasing the reliability, performance and scalability of the system (*What Is REST?*, 2021).

2.1.1 Client-server communication

Figure 2 shows how a client and server in a typical REST architecture communicate with each other. In this project, the Angular application serves as the client and the Spring Boot application as the server.

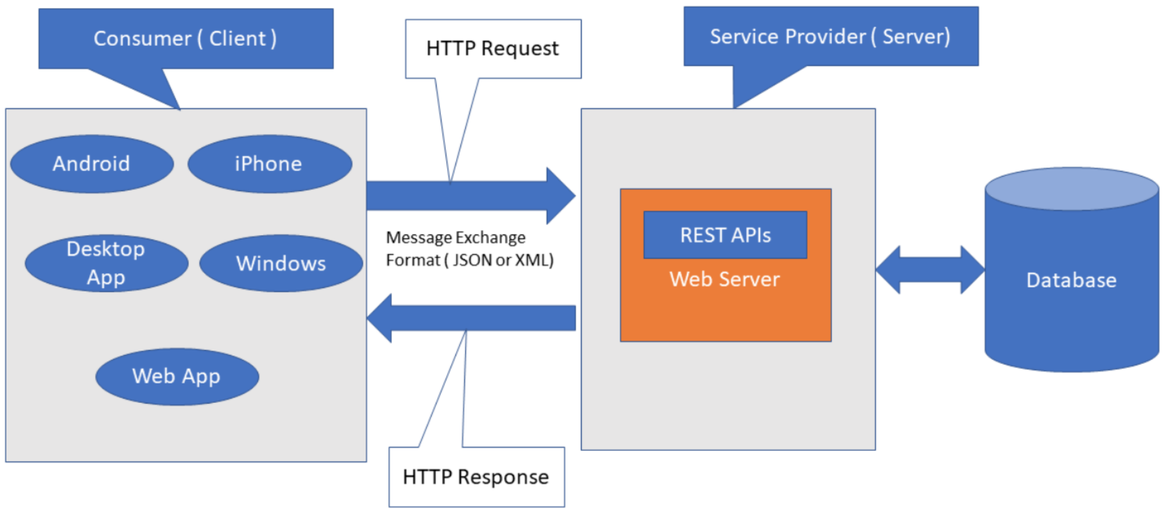


Figure 2. Client-server communication in a REST architecture (Fadatare, 2021).

In a REST architecture a client sends requests to the server to access or modify resources, and the server sends back responses. A resource could represent any type of object, data, or Service. Many web APIs use JSON as the message exchange format, but other formats such as XML can be used as well.

A request is composed of an HTTP method that defines what type of action to make, a unique path (URL) to a resource, a header with metadata, and an optional body with data. The most common HTTP methods are:

- GET - retrieves a resource or a collection of resources
- POST - creates a new resource
- PUT - updates a resource
- DELETE - deletes a resource

The resource path contains all the necessary data to locate a specific resource. It should be designed in a way that is easy to read and understand. Conventionally only nouns are used in the path. An example path for retrieving a specific user resource could look like: *http://api.example.com/users/12*. The last part is a path parameter, which indicates a user uniquely identified by the number 12.

When a server receives a request from a client it tries to fulfill the request by doing some operation that often involves modifying the database. It then sends back a response containing a status code and in some cases also a body with the requested resource. The most common status codes are (*What Is REST?*, 2021):

- 200 (OK) - the request was successful
- 201 (OK) - the request was created
- 204 (NO CONTENT) - the request was successful, but response body is empty
- 400 (BAD REQUEST) - the request cannot be processed due to some type of client error, such as bad request syntax
- 404 (NOT FOUND) - the requested resource could not be found
- 500 (INTERNAL SERVER ERROR) - the generic response if some unexpected failure happened while processing the request.

2.2 Model-View-Controller

Model-View-Controller (MVC) is a popular software design pattern that divides an application into three logical components. Those are the model, the view, and the controller (Bucanek, 2009):

- The model holds the data of the application and defines the logic that manipulates and processes that data.
- The view is the UI of an application and displays the data from the model.
- The controller functions as a mediator between the model and the view. It maps user input from the view to operations for modifying the model data. The controller then passes the updated data back to the view.

Figure 3 illustrates the MVC components and their interaction with each other.

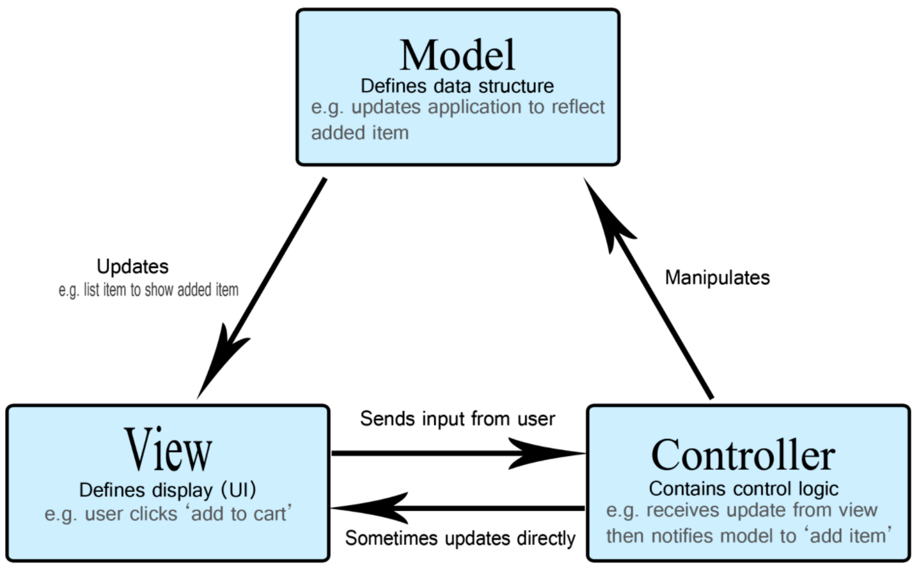

Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Figure 3. The interaction of the components in the MVC pattern (MVC - MDN Web Docs Glossary, 2021).

The purpose of the MVC pattern is to separate the presentation layer from the business logic, so that an application will be easier to manage and maintain (*MVC - MDN Web Docs Glossary*, 2021)).

2.3 IoC and dependency injection

Inversion of control (IoC) is a design principle, stating that objects should depend on abstractions (interfaces) for fulfilling specific tasks, not on the implementations themselves (*A Quick Intro to Dependency Injection: What It Is, and When to Use It*, 2018).

The dependency injection pattern is an implementation of IoC, where dependencies of an object are provided externally through a constructor or a setter, rather than the object constructing them itself. Dependencies are pieces of code that an object depends on for its functionality (*Angular*, n.d.-b).

Figure 4 presents an abstract example of how dependency injection works. The MovieLister class has a dependency to a MovieFinder interface. It does not instantiate the MovieFinder implementation directly. Instead, an Injector class creates the object and injects it into the MovieLister, ensuring that this class is totally unaware of how the dependency is created.

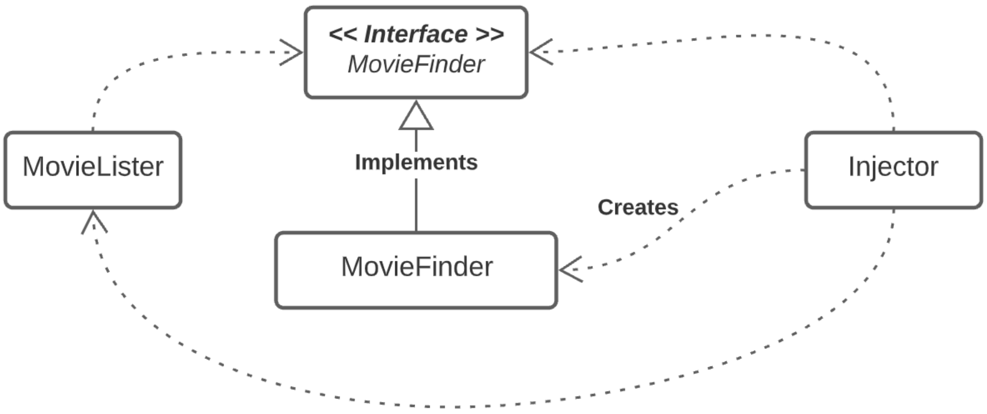


Figure 4. An example of dependency injection (Fowler, n.d.).

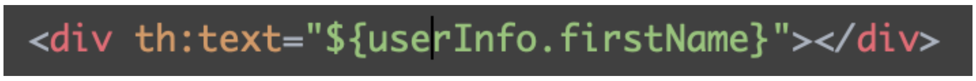
The purpose of dependency injection is to achieve loose coupling between objects so that they can be more easily extended and tested (*A Quick Intro to Dependency Injection: What It Is, and When to Use It*, 2018). Both Spring Boot and Angular are dependency injection frameworks.

3. FRONTEND TECHNOLOGIES

3.1 Thymeleaf

Thymeleaf is a server-side Java template engine for both web and standalone applications. It is able to process HTML, XML, JavaScript, CSS and plain text. It is commonly used for serving dynamic web pages (HTML) at the view layer of MVC-based Spring applications (see chapter 2.2 to learn more about the MVC architecture).

The idea behind template engines is to enforce separation between the application model and the view. Thymeleaf achieves this by using *attributes* and *variable expressions* to insert logic and data into template files. Variable expressions function as placeholders for the application data and they are evaluated by attributes. An attribute is associated with a specific HTML element, which it modifies based on the result of their expression values. Figure 5 is an example of how the attribute *th:text* is used to evaluate its value expression and replace the result with the text inside the HTML tag (*Tutorial: Using Thymeleaf*, n.d.).



```html
<div th:text="${userInfo.firstName}"></div>
```

Figure 5. Example of a Thymeleaf attribute used for replacing text in an HTML-tag.

3.2 Angular

Angular is an open-source platform and framework for developing single-page client applications using HTML and TypeScript (*Angular*, n.d.-a). A single page application is a web application that loads a page only once, and then updates the content of the page in response to user input (*SPA (single-Page Application)*, 2021).

Angular is written in TypeScript, which is a superset of JavaScript, which means TypeScript is essentially JavaScript but with additional features, such as interfaces and static typing (*Documentation - TypeScript for the New Programmer*, n.d.). Static typing means that the types of variable values are pre-defined (Meyer, 1996). With TypeScript's type system code errors get detected during compilation, before any code is running. This reduces runtime errors, which is a strong advantage Typescript has over JavaScript (*Documentation - TypeScript for the New Programmer*, n.d.).

3.2.1 Angular architecture

The Angular architecture is modular in its design. The purpose of this design is to subdivide the system into smaller units called modules, which can be developed independently and then assembled together. Modularity facilitates code reuse and improves manageability in that modules are easy to design, implement, test, and debug (*4.1 Modular Design Review*, n.d.).

The main building blocks in an Angular application are modules, components, templates, metadata, data binding, services and dependency injection. Figure 6 shows how these building blocks interact with each other.

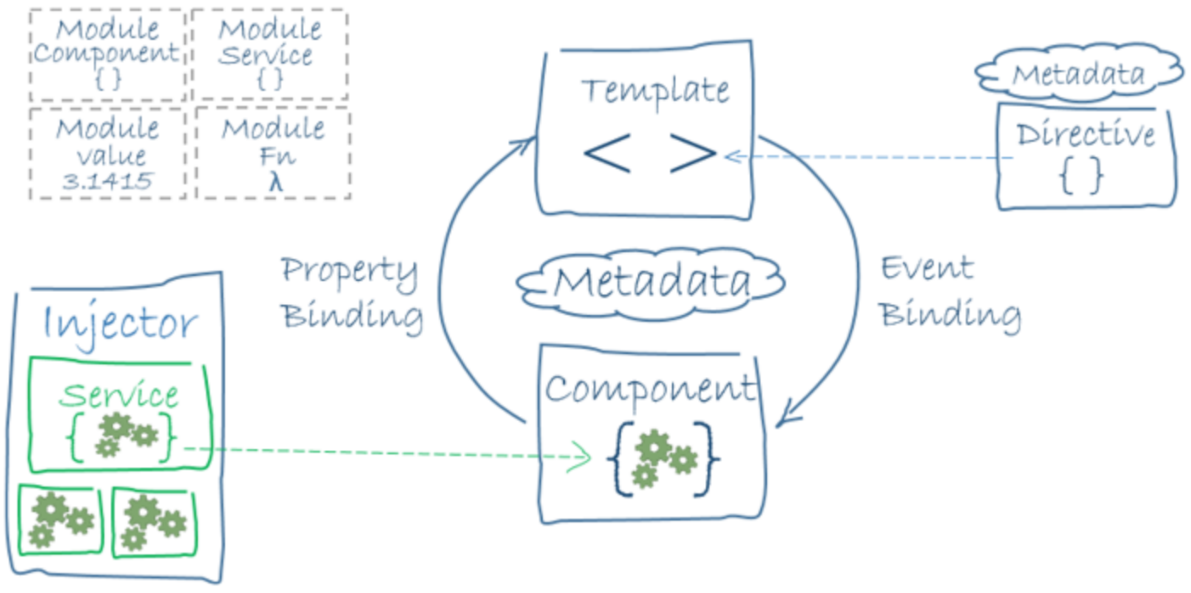
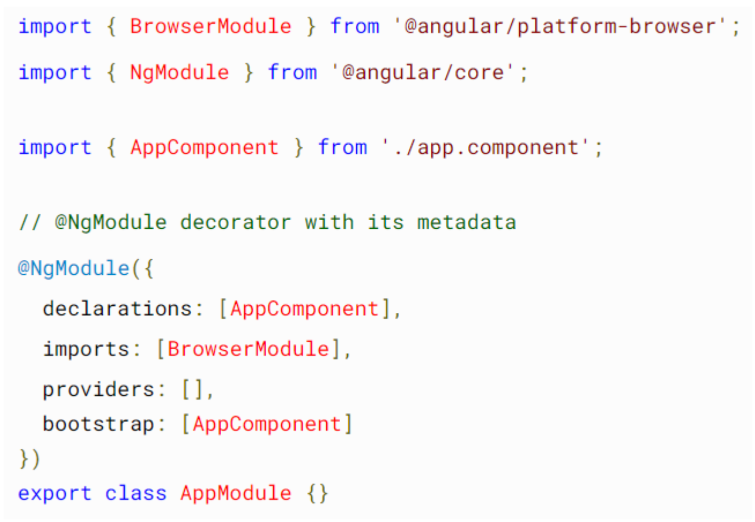


Figure 6. The building blocks in the Angular architecture (Angular, n.d.-a).

3.2.1.1 Modules

Angular has its own modularity system called NgModules. The purpose of NgModules is to collect components and/or services which are closely related into functional sets. Every Angular application has a root module, which launches the application. The root module can contain a hierarchy of child modules of any depth (*Angular*, n.d.-a). An example of a basic root module can be seen in figure 7.



```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

// @NgModule decorator with its metadata
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Figure 7. An example of a root module (Angular, n.d.-c).

An NgModule needs to include metadata describing which components and directives belong to it (declarations), as well as other modules it uses (imports)(*Angular*, n.d.-c). A root module also includes bootstrap metadata, that contains the root component that Angular creates and inserts into browser DOM when the application launches (*Angular*, n.d.-d).

3.2.1.2 Components and templates

Components are the main building blocks of an Angular application. A component is responsible for how a specific part of the UI looks and behaves. Components consist of:

- a component class, written in TypeScript, that contains application data and logic.
- a template that displays data provided by the component class. A template is HTML mixed with Angular markup that can modify HTML elements before they are displayed.
- optional style sheets that apply layout to the template.

The component class and the template communicate with each other through the concept of data-binding (*Angular*, n.d.-a).

3.2.1.3 Metadata

Metadata is used to provide Angular with instructions about how a class should be configured. Angular uses decorators to attach metadata inside classes. As an example, the @Component decorator marks a class as a component. The metadata for a component informs Angular where to get the resources needed to create and present the component and its view template. The component metadata usually includes a selector (used for referencing the component from a template), location of the template file, location of stylesheets, and what services the component requires (*Angular*, n.d.-e).

3.2.1.4 Data binding and directives

Binding markup connects application data with the DOM. There are two types of data binding: event binding and property binding. Event binding allows the application to respond to events raised by user actions, such as button clicks, by updating the application data. Property binding lets the component class insert application data into the template HTML.

Template directives are responsible for providing logic in a template. Angular evaluates the directives and modifies the DOM according to the instructions given by directives before the view gets displayed. There are three types of directives (*Angular*, n.d.-f):

1. Structural directives. Structural directives alter the layout of the template by modifying the structure of the DOM. For example, the *ngFor - directive iterates through a collection of objects.
2. Attribute directives. Attribute directives modify the behavior and look of DOM elements. An example of an attribute directive would be the *ngClass-directive, which adds or removes CSS classes for styling elements.
3. Components. These are directives with a template.

3.2.1.5 Services and dependency injection

Services are helper classes that provide functionality that can be used across multiple parts of the application. Components typically use services to execute specific tasks with narrow and well defined purposes. User input validation and server communication, such as fetching data, are examples of common tasks that should be delegated to a service. The idea behind using services is to increase modularity and reusability by keeping view-related logic in a component separate from other types of logic.

Services are made accessible to components by dependency injection. A service class is defined by using the *@injectable* decorator, which tells Angular to inject the service into a component's constructor as a dependency (*Angular*, n.d.-g). See chapter 2.3 to learn more about dependency injection.

3.2.1.6 Pipes

Pipes in Angular are functions used for changing the format in which data, such as strings and dates, is displayed. An advantage of pipes is that they only need to be declared once. Then they can then be used throughout the application (*Angular*, n.d.-h).

3.3 Server-side rendering versus client-side rendering

Since the aim for this project is to implement a change from server-side rendering (SSR) to client-side rendering (CSR), this chapter will go through the main difference of these two approaches and discuss some benefits and downsides of each approach.

SSR is the traditional way of serving web pages. When a request for a page is made, the server uses a template engine, such as Thymeleaf, to build a fully populated HTML page. This page is then sent to the client's browser to be displayed. Every time a user navigates to another URL, the server rebuilds the entire page from scratch.

CSR is a relatively modern way to serve web pages. With the CSR approach, the content of a a page is rendered dynamically on the browser as the user navigates a website. Instead of a full-blown HTML page, the server only renders a skeleton HTML container in the initial page load. The rest of the page rendering is handled by the browser using a JavaScript framework, which in this case is Angular. CSR applications make it possible to re-render only the parts of the UI that have updated content, which is much faster than having to re-render an entire page. Applications using this approach are called single page applications (SPA).

Below are listed the most important benefits and downsides of SSR and CSR (Vega, 2017). Benefits of SSR:

- The initial page load of the website is faster than CSR.
- Search engines are able to crawl the site for better SEO (Search Engine Optimization).

Downsides of SSR:

- Lots of server requests.
- Slow rendering if the website has a lot of interactivity
- Full page refresh for each request.

Benefits of CSR:

- Fast and responsive website.
- Many JavaScript frameworks and libraries support CSR.
- Clear separation of the client and the server. The code on the client side can be modified without affecting server functionality, and vice versa.

Downsides of CSR:

- Slower initial page load, due to all JavaScript that needs to be downloaded and executed before rendering page content.
- Good SEO can be hard to achieve due to load times and lack of initial HTML content.

4. FRAMEWORKS AND TOOLS

4.1. Spring Boot

Spring Boot is a Java-based framework for developing stand-alone, production-grade applications (*Spring Boot*, 2021). Spring Boot is built on top of the traditional Spring framework, which is a popular framework for creating Java enterprise applications. One of the main issues with Spring based applications is that they need a lot of configuration. In Spring Boot everything is auto-configured, which means that much less configuration and setup for the application is required. Autoconfiguration allows developers to focus more on the business logic, and it also reduces the risk of human error (*What Is Java Spring Boot*, 2020).

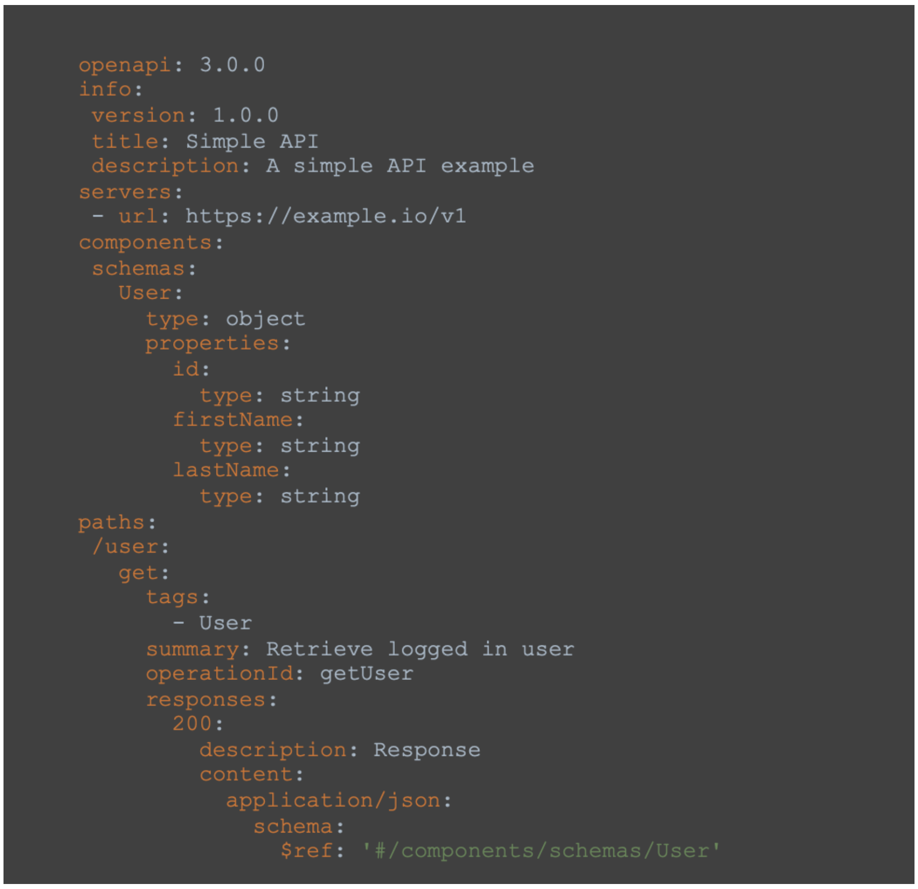
4.2. Swagger

Swagger is a set of tools that helps users design, document, test and consume REST APIs. The Swagger tools are built around OpenAPI specification (OAS), which is a standard interface for describing the structure of an API. The specification can be written in either YAML or JSON, and it is both machine and human readable. The specification typically contains information about the API's endpoints and their supported operations, as well as input and output parameters needed for each operation. It can also include information such as authentication methods, license and terms of use (*About Swagger Specification*, n.d.).

Swagger can be used with both top-down and bottom-up development approaches. In the top-down approach, the OAS is designed before any code is written. A code generator tool can then be used to create REST API interfaces. In the bottom up, or code first approach, Swagger generates the documentation automatically from the source code by asking the API to return a documentation file from its annotations (*Swagger API Documentation*, n.d.). For this project the design first method was applied.

4.2.1 OpenAPI specification example

Figure 8 is an example of a simple OpenAPI specification written in YAML.



```yaml
openapi: 3.0.0
info:
 version: 1.0.0
 title: Simple API
 description: A simple API example
servers:
 - url: https://example.io/v1
components:
 schemas:
   User:
     type: object
     properties:
       id:
         type: string
       firstName:
         type: string
       lastName:
         type: string
paths:
 /user:
   get:
     tags:
       - User
     summary: Retrieve logged in user
     operationId: getUser
     responses:
       200:
         description: Response
         content:
           application/json:
             schema:
               $ref: '#/components/schemas/User'
```

Figure 8. An example of a simple OpenAPI specification document.

In table 1 are given short descriptions of some of the specification properties from the example in figure 8.

Table 1. Description of properties from the example OpenAPI specification (About Swagger Specification, n.d.)

Property	***Description***
openapi	The current swagger version.
info	Metadata about the API, such as version, title and description.
servers	Contains the base URL for all endpoints.
components	Contains reusable objects, such as schemas, responses and headers, that are referenced from properties outside the components object.
schemas	Defines input and output data types which can be objects, primitives

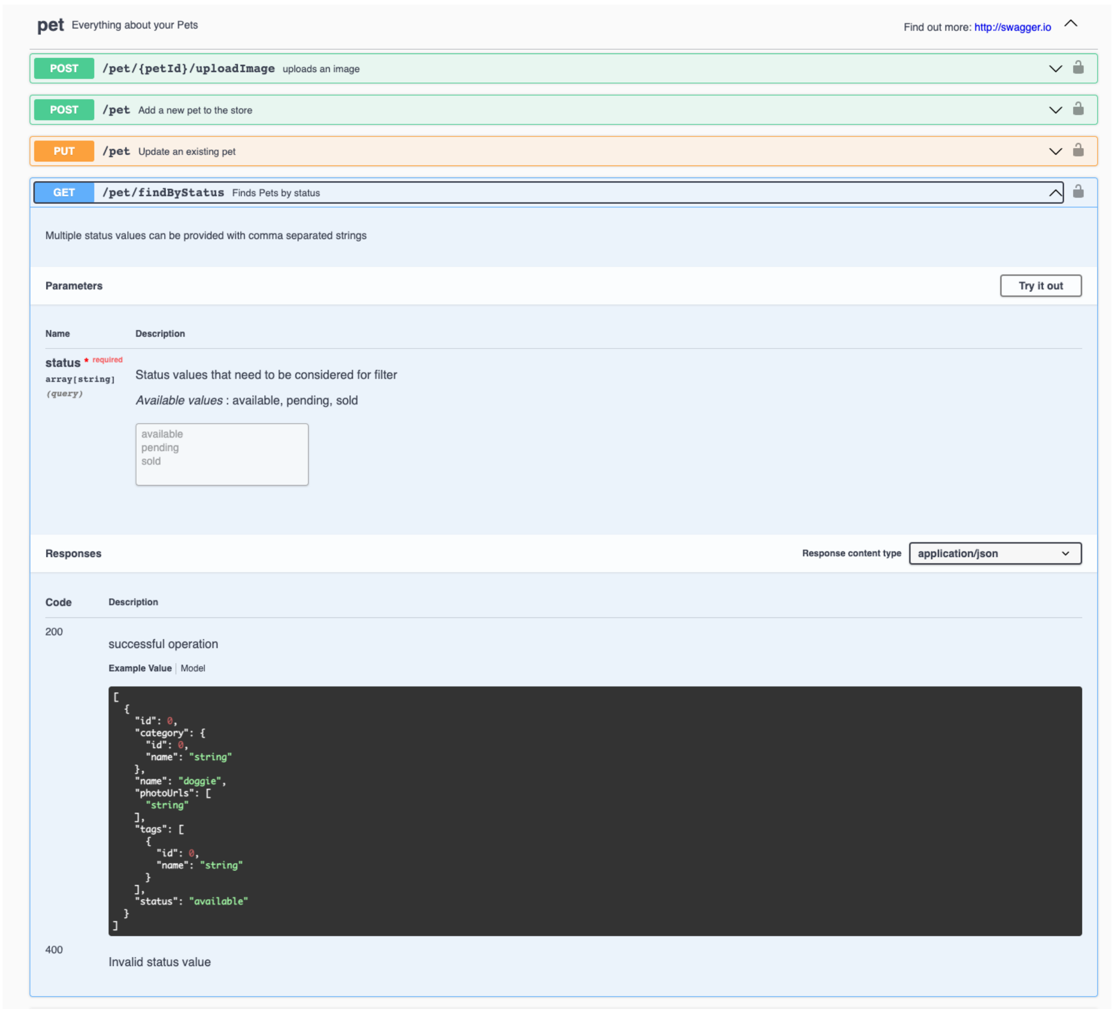
	and arrays. In the example a User object is defined, with fields for id, first name and last name.
paths	Holds the paths to the API’s endpoints and their respective operations. They are appended to the base URL defined in the server object. In this example the endpoint is /user and the full URL is https://example.io/v1/user. The operation (HTTP method) is GET, which is a request to read a specific resource. A path can contain parameters, which usually are included in the URL path (/user/{id}) or as query strings (/user?role=admin).
responses	Contains the responses of an operation. A response is mapped to at least one HTTP status code. In the GET /user request from the example the status code is 200, which indicates a successful operation. A response can contain a description, header, links and content (see below).
content	Contains the media types consumed by the operation, such as application/json. It also defines the schema for each media type. In the example the schema is the User object, which is referred to by using the $ref property.

4.2.2 OpenAPI Generator

OpenAPI Generator is a tool that can generate client and server side code based on the OAS (*OpenAPI Generator*, 2021). OpenAPI Generator originates from Swagger Codegen. The main difference is that Swagger Codegen is driven by SmartBear while OpenAPI Generator is community driven (*OpenAPI Generator FAQ*, 2021).

4.2.3 Swagger UI

Swagger UI automatically generates visual and interactive documentation from the OAS. In the generated document a user can get a quick overview of the API and try out its operations (*Swagger UI*, 2021). An example of how OAS documentation is presented with Swagger UI can be seen in figure 9.


pet Everything about your Pets

Find out more: http://swagger.io

POST /pet/{petId}/uploadImage uploads an image

POST /pet Add a new pet to the store

PUT /pet Update an existing pet

GET /pet/findByStatus Finds Pets by status

Multiple status values can be provided with comma separated strings

Parameters

Try it out

Name	Description
status * required array[string] (query)	Status values that need to be considered for filter *Available values* : available, pending, sold available pending sold

Responses

Response content type: application/json

Code	Description
200	successful operation
400	Invalid status value

Example Value | Model

```
[
  {
    "id": 0,
    "category": {
      "id": 0,
      "name": "string"
    },
    "name": "doggie",
    "photoUrls": [
      "string"
    ],
    "tags": [
      {
        "id": 0,
        "name": "string"
      }
    ],
    "status": "available"
  }
]
```

Figure 9. An example of how the documentation for an API looks like with Swagger UI (Swagger UI, 2021).

4.3 Selenium

Selenium is an open source automated testing suite for web applications and can be used by any browser or platform. It includes Selenium WebDriver, which is a tool that can be used to create and execute test cases for a web application UI. Tests are created using locators to identify HTML elements to then carry out actions on those elements, such as opening a web page, clicking a button, etc. HTML elements can for instance be located by its class name, as shown in figure 10 below.

```
WebElement vegetable = driver.findElement(By.className("tomatoes"));
```

Figure 10. A Java example of the mechanism Selenium uses for locating an HTML element by its class name (Finding Web Elements, n.d.).

Selenium test scripts can be written in several different programming languages, but the one used ín this case was Java. The test scripts interact directly with the browser, driving the browser just like an actual user would do (*WebDriver*, 2021). The test scripts for this project were configured to run in the Chrome browser.

5. APPLICATION DEVELOPMENT

5.1 Preparation

Before starting with the application development, I used JIRA as a tool to structure and to document my work on the project. I could split my work into smaller and more manageable tasks by writing JIRA stories that described what needed to be done for each task.

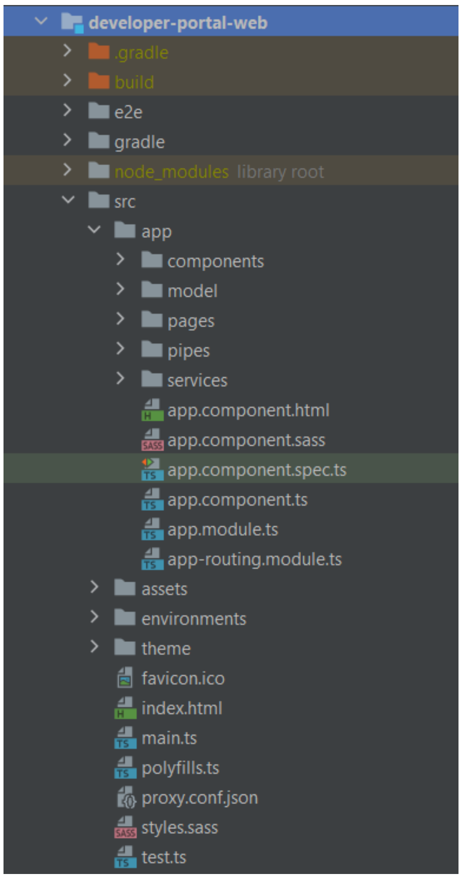
There was no need for me to create an Angular application completely from scratch, since the project of converting the frontend into Angular had been started earlier by other developers in my team. The only thing I had to do to get started with the frontend development was to install the Angular CLI[2] on my work computer. Angular CLI is a command-line interface tool that comes with a set of commands to facilitate the development of Angular applications. It was particularly useful in this project for creating new components and services. For instance, to create a new component you type the command "ng generate component <component-name>", and the Angular CLI will create a folder containing all the required files, and add the necessary configurations.

5.2 Frontend development

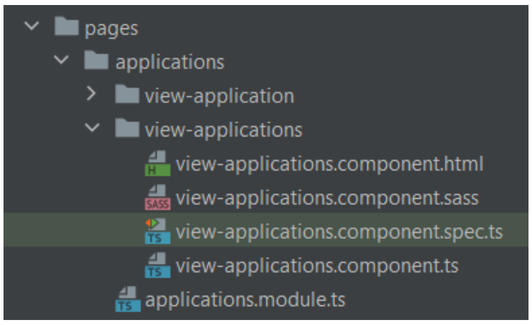
5.2.1 Application file structure

An overview of the file structure in the Angular application are presented in figures 11 and 12 below.

[2] https://angular.io/cli



```
developer-portal-web
  > .gradle
  > build
  > e2e
  > gradle
  > node_modules  library root
  v src
    v app
      > components
      > model
      > pages
      > pipes
      > services
        app.component.html
        app.component.sass
        app.component.spec.ts
        app.component.ts
        app.module.ts
        app-routing.module.ts
    > assets
    > environments
    > theme
      favicon.ico
      index.html
      main.ts
      polyfills.ts
      proxy.conf.json
      styles.sass
      test.ts
```



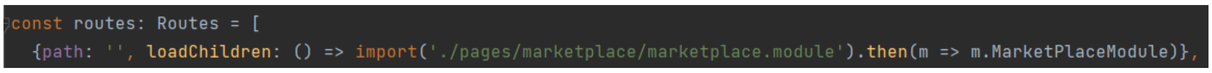
```
v pages
  v applications
    > view-application
    v view-applications
        view-applications.component.html
        view-applications.component.sass
        view-applications.component.spec.ts
        view-applications.component.ts
      applications.module.ts
```

Figure 11 and 12. The file structure of the Angular application.

The work for this project mainly involved adding or modifying files placed in the *app* folder. This folder include:

- pages folder, containing the components that make up a web page.
- components folder, containing reusable components that are used throughout the application, such as header, footer, etc.
- model folder, containing the interfaces for mapping resource objects from the backend.
- pipes folder, containing the pipes used throughout the application.
- services folder, containing the service classes used by the page components.
- app.module.ts. The application launches by bootstrapping this module.
- the root component, which represents the entry point of the application. All other components will branch off from this component. The root component consists of:
  - app.component.ts - the component class defining the view logic.

    - app.component.html - the HTML template. This file includes elements that are present in all pages, such as the header and the footer. An important element to include in this file as well is *<router-outlet>*. This element acts as a placeholder, that Angular dynamically fills with a page component based on the selected route (URL)..
    - appt.component.spec.ts - unit tests for the component are written in this file.
    - app.component.sass - the stylesheet of the component. All stylesheets in this application are written in SASS format. SASS is compiled to CSS, but compared to CSS, it has more features. It introduces variables, functions and nested rules, among other things (*Documentation*, n.d.).
- app-routing.module.ts. This file handles the navigation for the web site. Figure 13 shows how app-routing.module.ts has been implemented in this application. The *loadChildren()* function is used for lazy loading of modules. Lazy loading means a module will load only when it's needed. This results in a smaller initial bundle size, which helps to achieve a faster initial load time (*Angular*, n.d.-i).

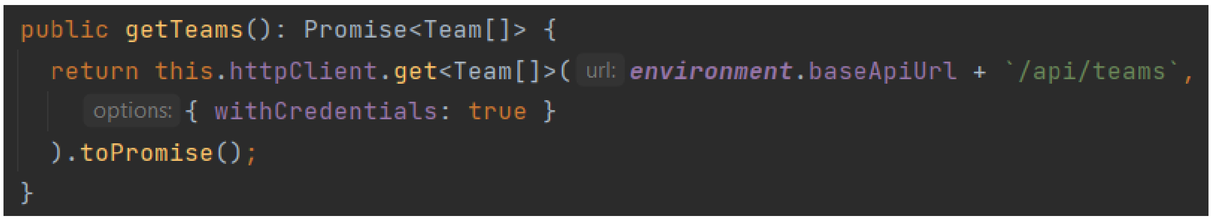


```
const routes: Routes = [
  {path: '', loadChildren: () => import('./pages/marketplace/marketplace.module').then(m => m.MarketPlaceModule)},
```

Figure 13. Routing example with lazy loading.

5.2.2 Backend communication

To be able to communicate with the server using the HTTP protocol, I used Angular's HTTP API, the service class *HttpClient*. This API offers several useful features, among which are the ability to set the types of the response objects, and to intercept the requests and responses (*Angular*, n.d.-j). Figure 14 shows an example of a service method, *getTeams()*, that fetches data from the server. The expected response from the request is an array of Team objects.



```
public getTeams(): Promise<Team[]> {
  return this.httpClient.get<Team[]>( url: environment.baseApiUrl + `/api/teams`,
     options: { withCredentials: true }
  ).toPromise();
}
```

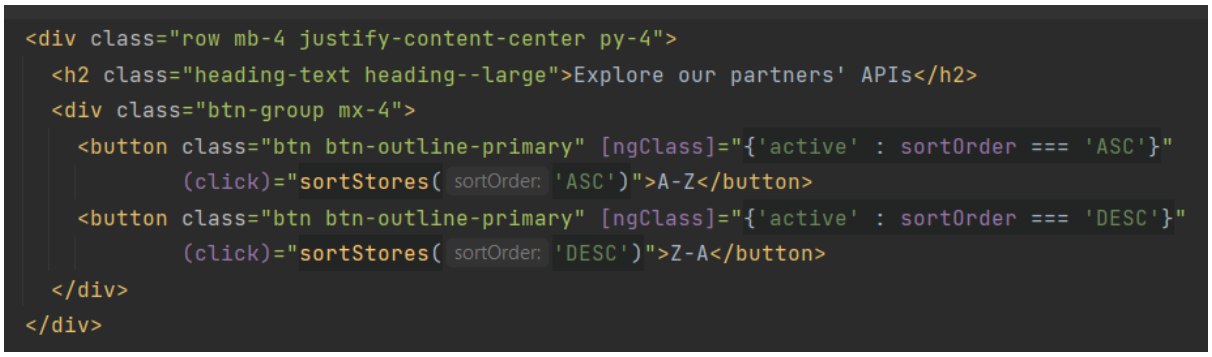
Figure 14. Server request with httpClient.

As mentioned earlier, HttpClient offers the possibility to intercept HTTP requests and responses. Interceptors can be used to catch and handle errors that might arise while making requests to the server. Angular's *HttpInterceptor* interface is used for handling HTTP errors globally in the application. The idea is to get consistent error handling by handling common errors in one single place, and only let specific errors be handled in services or components.

To achieve global error handling I created an error interceptor service class that implements the HttpInterceptor-interface and registered it as a provider in the root module. Whenever a request fails on the backend, HttpClient returns an error object. The error interceptor's handle method catches this error and handles it differently based on what type of error it is. For instance, if it's an error with status code 404, meaning the requested resource couldn't be found, the interceptor will redirect the user to an page that displays the status code of the error as well as an appropriate message informing the user what went wrong.

5.2.3 Handling user input

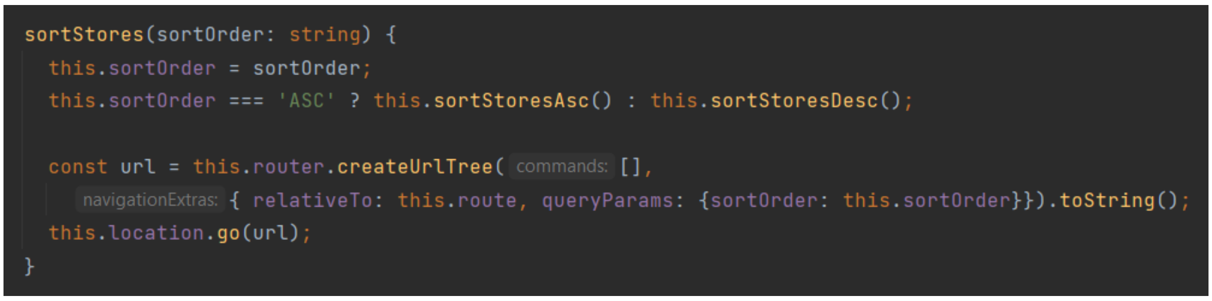
Handling of user input can be done efficiently in Angular by using features like event binding. DOM elements in the view template can be set to listen for user events, such as keystrokes, button clicks etc, to then trigger a specific action when an event is detected. The Angular syntax for binding to an event consists of a target event name within parentheses, followed by an equal sign and a quoted template statement. An example of event binding is shown in figure 15. This piece of template code represents the section of the UI for changing the sort order of store items in the home page. The buttons in the figure listens for click events, using the *(click)* target event. Whenever a button is clicked, the *sortStores()* method defined in the component class will get called.



```
<div class="row mb-4 justify-content-center py-4">
  <h2 class="heading-text heading--large">Explore our partners' APIs</h2>
  <div class="btn-group mx-4">
    <button class="btn btn-outline-primary" [ngClass]="{'active' : sortOrder === 'ASC'}"
            (click)="sortStores( sortOrder: 'ASC')">A-Z</button>
    <button class="btn btn-outline-primary" [ngClass]="{'active' : sortOrder === 'DESC'}"
            (click)="sortStores( sortOrder: 'DESC')">Z-A</button>
  </div>
</div>
```

Figure 15. Example of event binding in Angular template.

In figure 16 can be seen how the logic for sorting store items is implemented in the component class. This function adds the *sortOrder* variable as a query parameter to the current url. The *location.go()* function is used for changing the current URL to the given URL without reloading the page.



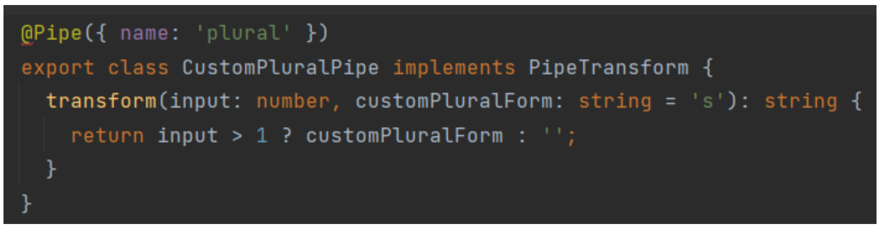
```
sortStores(sortOrder: string) {
  this.sortOrder = sortOrder;
  this.sortOrder === 'ASC' ? this.sortStoresAsc() : this.sortStoresDesc();

  const url = this.router.createUrlTree( commands: [],
    navigationExtras: { relativeTo: this.route, queryParams: {sortOrder: this.sortOrder}}).toString();
  this.location.go(url);
}
```

Figure 16. Implementation of method for sorting stores in Angular component..

With event binding Angular can handle a lot of the logic for user interaction in the browser. This is an improvement from the previous Thymeleaf UI, where almost all types of user input was handled by sending HTTP requests to the server using forms, resulting in full page reloads.

5.2.4 Using pipes

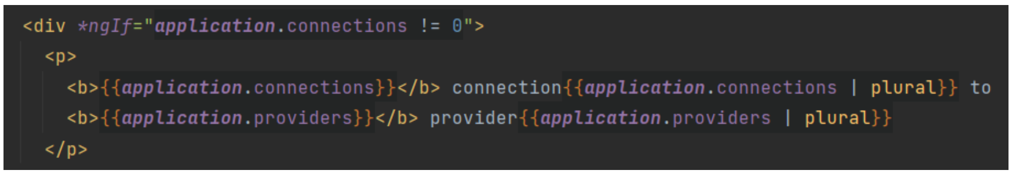
Angular supports a lot of built-in pipes. One of these is DatePipe, which I used in the application to transform dates into a specific format. It's also possible to create your own custom pipes that suit your requirements. For this application I built a simple custom pipe, seen in figure 17, that would make plurals of words by taking a number as input and return the letter "s" if the number is larger than 1.



```
@Pipe({ name: 'plural' })
export class CustomPluralPipe implements PipeTransform {
  transform(input: number, customPluralForm: string = 's'): string {
    return input > 1 ? customPluralForm : '';
  }
}
```

Figure 17. Implementation of a pipe class.

To refer to a pipe in the templates, a pipe operator (|) is used along with the name of the pipe. An example of how the plural-pipe is used is shown in figure 18.



```
<div *ngIf="application.connections != 0">
  <p>
    <b>{{application.connections}}</b> connection{{application.connections | plural}} to
    <b>{{application.providers}}</b> provider{{application.providers | plural}}
  </p>
```

Figure 18. Usage of a pipe in a template.

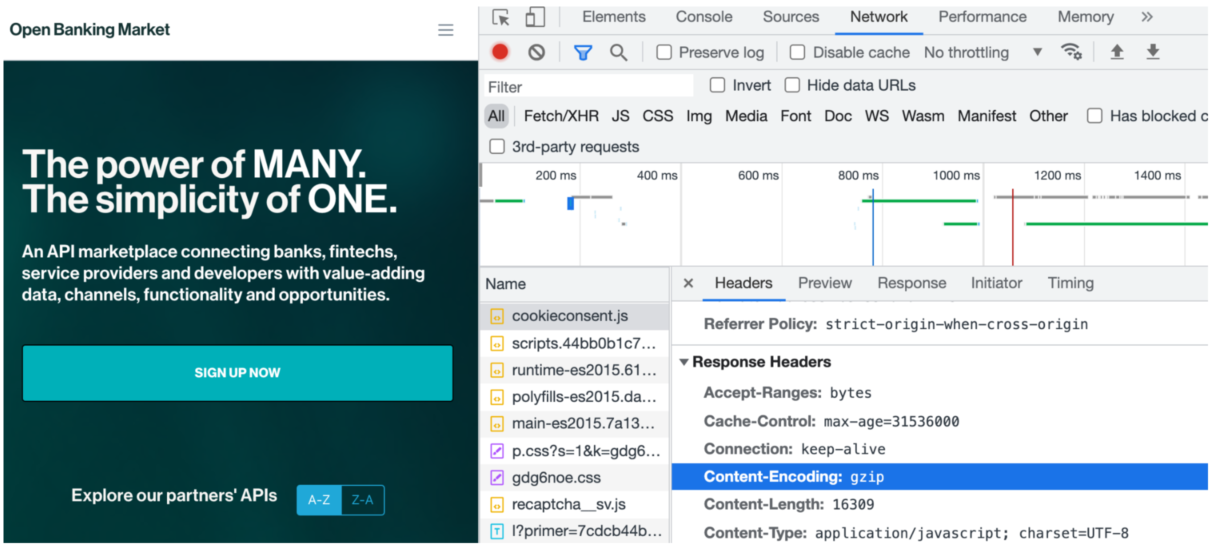
In the previous UI, the logic for formatting data was added directly in the template. With the use of pipes, this type of logic could be removed, resulting in cleaner and more readable templates.

5.3 Angular file compression

When the first two pages of the web platform had been converted to Angular, the performance of the new UI was measured using a Chrome developer tool called Lighthouse. Lighthouse audits a web page for performance, accessibility, SEO, among other things. It will generate a report that includes a score, with a s 0-100 scale grading, on how well the page did and suggestions for improvement. According to the report the performance had degraded somewhat from the old version of the UI. This was mainly due to the fact that the initial page load time (i.e the time it takes for a page to launch the first time) was slower than before. The report suggested that a faster load time could be achieved by using file compression to reduce the size of the bundle containing all the JavaScript files that the Angular application needs to download, parse and execute.

To enable file compression in the Angular application, I installed a file compression tool called Gzipper. Then I configured Gzipper to automatically run after an Angular build has been completed. A folder with all compiled resources will then be generated, containing all the compressed files as well as the original files. Since the resource files will be served by the backend, I had to configure the Spring Boot application to use the compressed files instead of the originals.

I could verify that the website uses gzipped files by checking the network tab in the browser's developer console. If a file is Gzipped, the response header "Content-Encoding" should display the value "gzip" as shown in figure 19.



cookieconsent.js
scripts.44bb0b1c7…
runtime-es2015.61…
polyfills-es2015.da…
main-es2015.7a13…
p.css?s=1&k=gdg6…
gdg6noe.css
recaptcha__sv.js
l?primer=7cdcb44b…

```
Referrer Policy: strict-origin-when-cross-origin
▼ Response Headers
  Accept-Ranges: bytes
  Cache-Control: max-age=31536000
  Connection: keep-alive
  Content-Encoding: gzip
  Content-Length: 16309
  Content-Type: application/javascript; charset=UTF-8
```

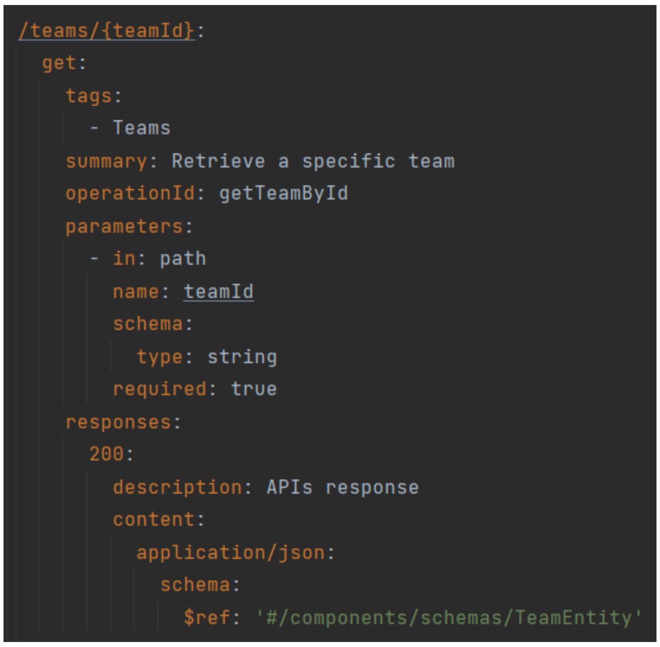
Figure 19. The response header "Content-Encoding: Gzip" indicates that a JavaScript file used in the crosskey.io website has been compressed to Gzip format.

After file compression the Lighthouse performance score increased with about 20 points, which is an improvement of approximately 30 percent. An exact number is not possible to give, since the score often varied between different runs. The lowest score that I measured was 56, and the highest was 85.

5.4 Design and implementation of the REST API

As described in chapter 2.1, REST APIs are used for communication between the client and the server. I used the top down method, also known as design first method, for developing the REST API. This means designing the API contract first, before writing any code. The contract was documented using the OpenAPI Specification (OAS). An OAS document written in YAML already existed in the project, so I had only to expand this document with new data. To get a sense of which endpoints, operations and resources (data objects) the API would need, I studied the code of the controller and model classes, as well as view templates from the previous MVC architecture.

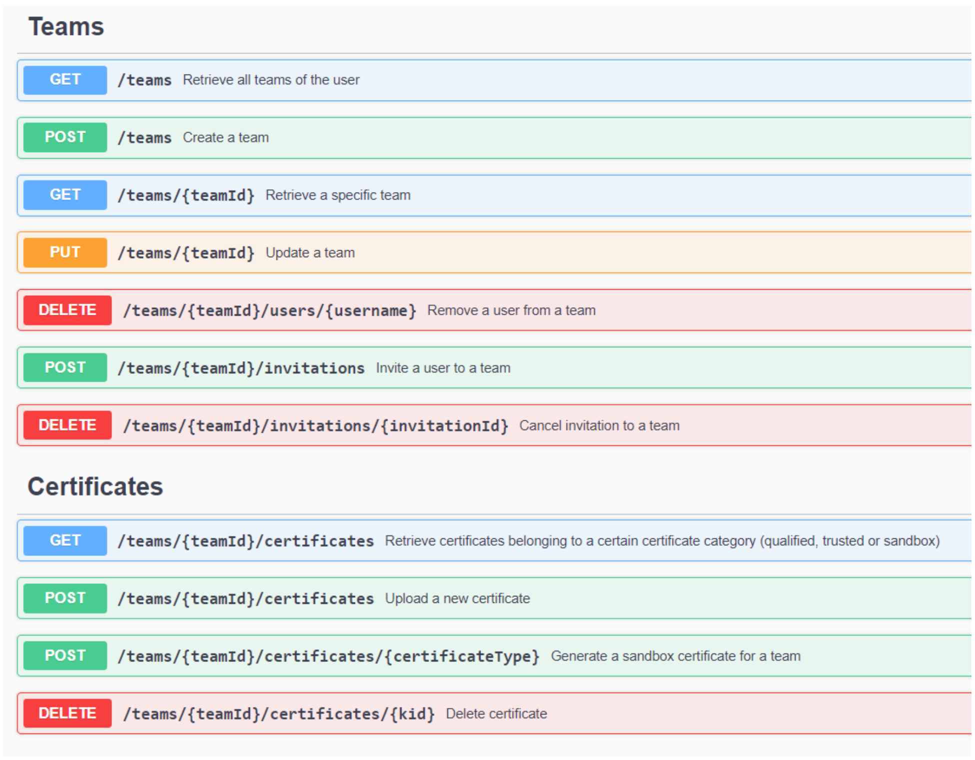
In figure 20 can be seen how the request, *GET /teams/{teamId},* was defined in the OAS. The *tags* property defines a category for the request, which is named Teams. It is used by Swagger UI to group operations of the same category. The URL path contains a required parameter, teamId, which has a string as its data type. The *response* property defines the operation's response, which in this case is successful with the status code 200. The response contains a resource object called TeamEntity, which is referred to by the *$ref* property. The application/json property indicates the resource object format is JSON. More information about the OAS can be found in chapter 4.2.



```yaml
/teams/{teamId}:
  get:
    tags:
      - Teams
    summary: Retrieve a specific team
    operationId: getTeamById
    parameters:
      - in: path
        name: teamId
        schema:
          type: string
        required: true
    responses:
      200:
        description: APIs response
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TeamEntity'
```

Figure 20. Definition of a request in the OAS document

The OAS is documented visually by using Swagger UI. In figure 21 are seen all available operations and endpoints for team resources.



Teams

GET /teams Retrieve all teams of the user

POST /teams Create a team

GET /teams/{teamId} Retrieve a specific team

PUT /teams/{teamId} Update a team

DELETE /teams/{teamId}/users/{username} Remove a user from a team

POST /teams/{teamId}/invitations Invite a user to a team

DELETE /teams/{teamId}/invitations/{invitationId} Cancel invitation to a team

Certificates

GET /teams/{teamId}/certificates Retrieve certificates belonging to a certain certificate category (qualified, trusted or sandbox)

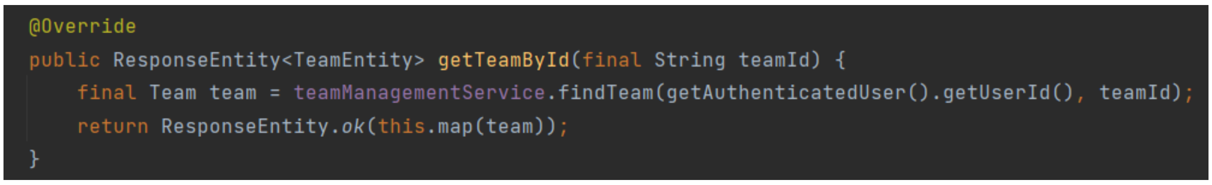
POST /teams/{teamId}/certificates Upload a new certificate

POST /teams/{teamId}/certificates/{certificateType} Generate a sandbox certificate for a team

DELETE /teams/{teamId}/certificates/{kid} Delete certificate

Figure 21. Some of the API documentation displayed with Swagger UI.

Using the OpenAPI generator tool, Spring Boot builds Java interfaces and model classes based on the requests and resources defined in the OAS document. One interface per main resource, such as teams and applications, were generated. The next step was to write REST controllers that implement these interfaces. By annotating the controller classes with @RestController, the class is marked as a request handler where every request handling method returns a *ResponseEntity* object. This represents an HTTP response, including status code, headers and body. Figure 22 shows the implementation of the *GET /teams/{teamId}* request seen defined in figure 20. This simple method uses a service to fetch a team object from the application model. The team object is then mapped to an API resource, *TeamEntity.* The method returns this data with the status code set to OK (200).



```java
@Override
public ResponseEntity<TeamEntity> getTeamById(final String teamId) {
    final Team team = teamManagementService.findTeam(getAuthenticatedUser().getUserId(), teamId);
    return ResponseEntity.ok(this.map(team));
}
```

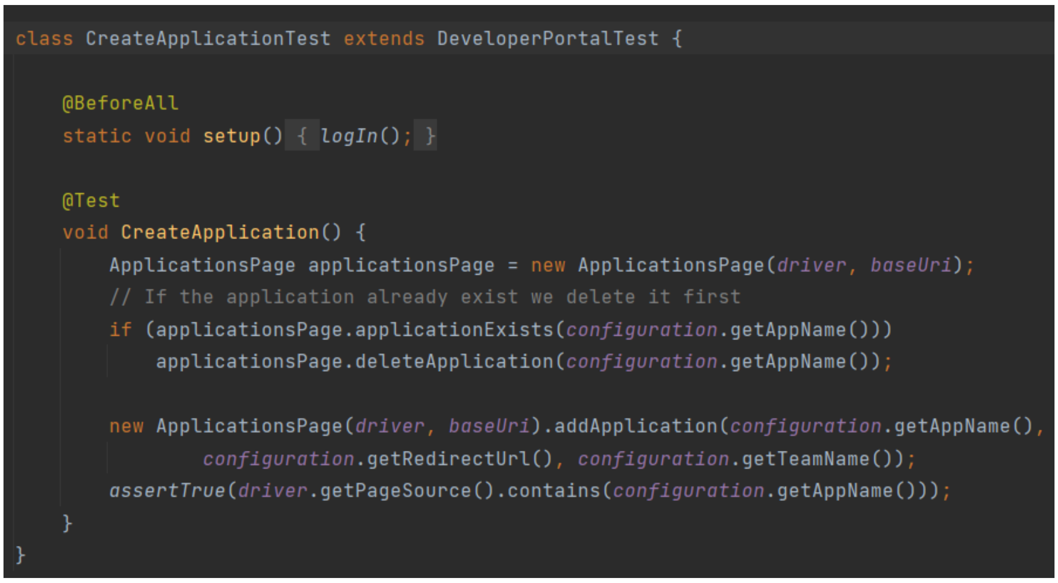
Figure 22. Method in a REST controller class for retrieving a team resource.

The service classes were injected into the REST controllers using dependency injection. By annotating a controller constructor with *@Autowired,* the service classes that are passed to the constructor are automatically instantiated.

5.5 Testing

5.5.1 End-to-end testing with Selenium

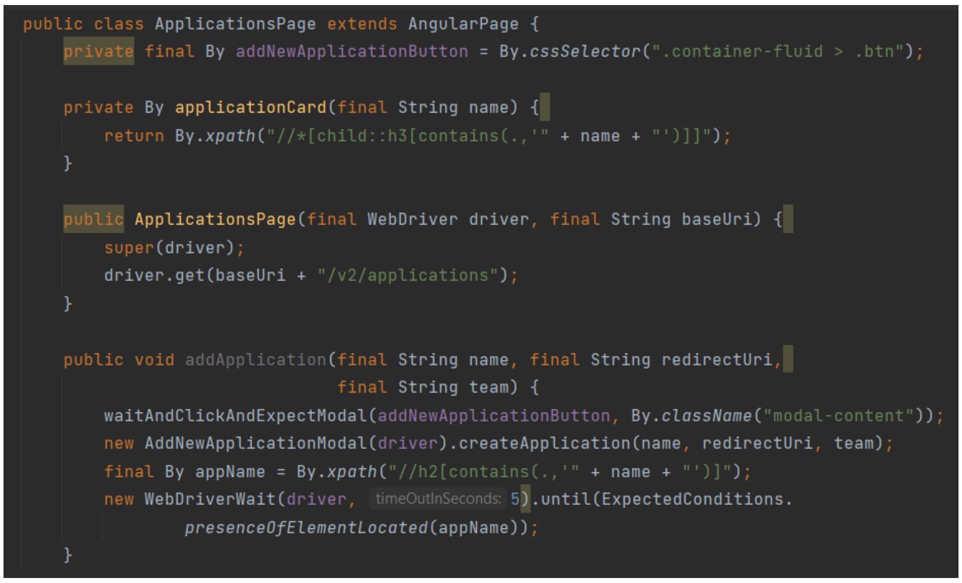
Selenium testing was used for the end-to-end testing of this website's UI. This means the functionality of the website is tested from a user perspective to make sure that the application flow is working as expected. Figure 23 shows a sample Selenium script that tests the user scenario of creating an application. When this test is run, the WebDriver will first log in the test user. Next, it navigates to the applications page. If the application already exists, it will get deleted. When an application has been added, a redirect is made to the page with details about the newly created application. The test will succeed if the application name can be found on that page.



```java
class CreateApplicationTest extends DeveloperPortalTest {

    @BeforeAll
    static void setup() { logIn(); }

    @Test
    void CreateApplication() {
        ApplicationsPage applicationsPage = new ApplicationsPage(driver, baseUri);
        // If the application already exist we delete it first
        if (applicationsPage.applicationExists(configuration.getAppName()))
            applicationsPage.deleteApplication(configuration.getAppName());

        new ApplicationsPage(driver, baseUri).addApplication(configuration.getAppName(),
                configuration.getRedirectUrl(), configuration.getTeamName());
        assertTrue(driver.getPageSource().contains(configuration.getAppName()));
    }
}
```

Figure 23. A Selenium test for creating an application.

A common reason for test failure was that the test scripts tried to locate HTML elements in an Angular page before the page was fully loaded. I solved this by adding *wait* methods that suspends execution of the script until a certain condition has been fulfilled. For instance, to make the test in figure 23 run successfully, I added a wait statement in the *addApplication*-method directly after a new application has been created. This will pause the execution of the test until the page with application details loads its content and the HTML element holding the name of the new application is found. See figure 24.

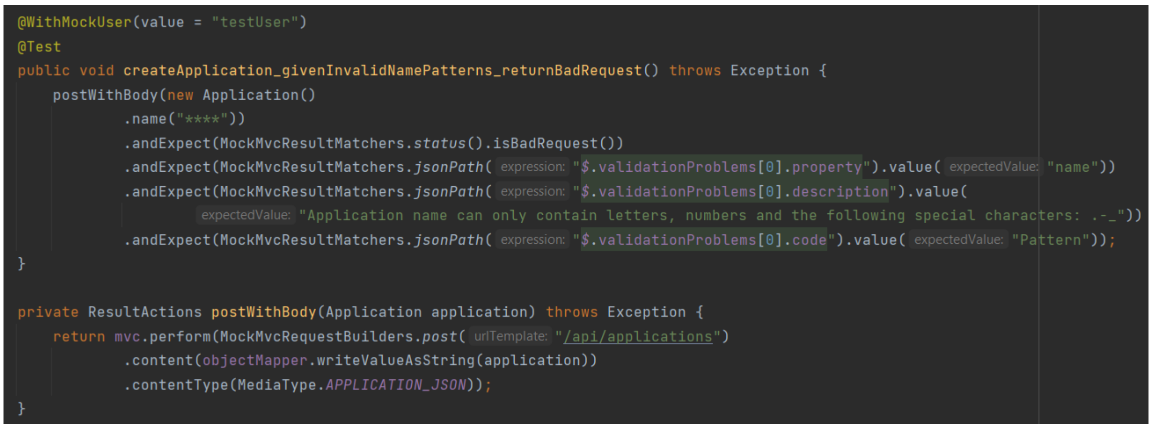


```java
public class ApplicationsPage extends AngularPage {
    private final By addNewApplicationButton = By.cssSelector(".container-fluid > .btn");

    private By applicationCard(final String name) {
        return By.xpath("//*[child::h3[contains(.,'" + name + "')]]");
    }

    public ApplicationsPage(final WebDriver driver, final String baseUri) {
        super(driver);
        driver.get(baseUri + "/v2/applications");
    }

    public void addApplication(final String name, final String redirectUri,
                               final String team) {
        waitAndClickAndExpectModal(addNewApplicationButton, By.className("modal-content"));
        new AddNewApplicationModal(driver).createApplication(name, redirectUri, team);
        final By appName = By.xpath("//h2[contains(.,'" + name + "')]");
        new WebDriverWait(driver,  timeOutInSeconds: 5).until(ExpectedConditions.
                presenceOfElementLocated(appName));
    }
```

Figure 24. The implementation of the addApplication-method for the applications page.

5.5.2 API testing

Some tests were written for the REST Controller classes on the backend to verify that API requests returned the correct responses. An example test is shown in figure 25. The test is checking that a request to create a new application object will return a 400 error (Bad Request) if the application name is invalid. The *postWithBody* method is a helper method that makes a mock API request and returns the response. A method, *andExpect*, can then be used to extract content from the response and compare it with the expected values.



```java
@WithMockUser(value = "testUser")
@Test
public void createApplication_givenInvalidNamePatterns_returnBadRequest() throws Exception {
    postWithBody(new Application()
            .name("****"))
            .andExpect(MockMvcResultMatchers.status().isBadRequest())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.validationProblems[0].property").value( expectedValue: "name"))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.validationProblems[0].description").value(
                     expectedValue: "Application name can only contain letters, numbers and the following special characters: .-_"))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.validationProblems[0].code").value( expectedValue: "Pattern"));
}

private ResultActions postWithBody(Application application) throws Exception {
    return mvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/api/applications")
            .content(objectMapper.writeValueAsString(application))
            .contentType(MediaType.APPLICATION_JSON));
}
```

Figure 25. An API test for checking that a request to create a new application with an invalid name returns an error response.

6. CONCLUSION

6.1. Result

The purpose of this project was to migrate the UI of the web application *crosskey.io,* from server-side to client-side, with the end goal to improve the performance of the web application. Since the focus lay on converting the most commonly used pages, I did not have time to convert the nice-to-have pages within the time frame for this project. The must-have pages in the requirement specification listed in chapter 1.3 have, however, been converted into Angular and they have all been deployed to the production environment.

There are bottlenecks on the server side that make some server calls take quite a long time to execute, and this is something that should be improved in the future. The overall performance and speed of the new client-side UI has, however, been noticeably improved compared with the previous server-side UI. Dynamic content on the website loads fast and seamlessly, without annoying “page blinks” due to page reloads.

6.2. Reflections

Working on this project has been both interesting and enjoyable. The difficulty level was reasonable and I have gained deeper knowledge of quite a few technologies and frameworks. Above all, I had the opportunity to expand my knowledge of Angular and TypeScript, which were my personal goals for this project.

One of the things I found a bit challenging with this project was making Selenium tests work with Angular. There were issues with synchronization and Selenium not having Angular-specific locating strategies, among other things. A consideration for the future would be to use another testing framework more suitable for Angular, such as Cypress or Protractor.

Another consideration for the future when the UI has been fully migrated to Angular, is to deploy the application in the cloud using AWS, which would bring benefits like improved scalability and reliability.

REFERENCES

4.1 Modular Design Review. (n.d.). https://www.mcs.anl.gov/~itf/dbpp/text/node40.html

About Crosskey. (2015, January 12). https://www.crosskey.fi/our-story//

About Swagger Specification. (n.d.). https://swagger.io/docs/specification/about/

Angular. (n.d.-a). https://angular.io/guide/architecture

Angular. (n.d.-b). https://angular.io/guide/dependency-injection

Angular. (n.d.-c). https://angular.io/guide/ngmodules

Angular. (n.d.-d). https://angular.io/guide/bootstrapping

Angular. (n.d.-e). https://angular.io/guide/architecture-components

Angular. (n.d.-f). https://angular.io/guide/glossary

Angular. (n.d.-g). https://angular.io/guide/architecture-services

Angular. (n.d.-h). https://angular.io/guide/pipes

Angular. (n.d.-i). https://angular.io/guide/lazy-loading-ngmodules

Angular. (n.d.-j). https://angular.io/guide/http

A quick intro to Dependency Injection: what it is, and when to use it. (2018, October 18). freeCodeCamp.org. https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

Atlassian. (n.d.). *Jira*. Atlassian. https://www.atlassian.com/software/jira

Bucanek, J. (Ed.). (2009). Model-View-Controller Pattern. In *Learn Objective-C for Java Developers* (pp. 353–402). Apress.

Davis, A. M. (1992). Operational prototyping: a new development approach. *IEEE Software*, *9*(5), 70–78.

Documentation. (n.d.). https://sass-lang.com/documentation

Documentation - TypeScript for the New Programmer. (n.d.).

https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html

Fadatare, R. (2021, May 15). *REST API Tutorial*. https://www.javaguides.net/p/rest-api-tutorial.html

Finding web elements. (n.d.). Selenium. https://www.selenium.dev/documentation/webdriver/elements/finders/

Fowler, M. (n.d.). *Inversion of Control Containers and the Dependency Injection pattern*. Martinfowler.com. https://martinfowler.com/articles/injection.html

Introduction to the DOM. (n.d.). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

JSON. (n.d.). https://www.json.org/json-en.html

Krause, J. (2016). HTML: Hypertext markup language. In *Introducing Web Development* (pp. 39–63). Apress.

Meyer, B. (1996). Static typing. In *Lecture Notes in Computer Science* (pp. 57–75). Springer Berlin Heidelberg.

MVC - MDN web docs glossary. (2021). https://developer.mozilla.org/en-US/docs/Glossary/MVC

OpenAPI generator. (2021). https://openapi-generator.tech/

OpenAPI generator FAQ. (2021). https://openapi-generator.tech/docs/faq

Open Banking Market-Crosskey. (n.d.). *Open Banking Market*. https://crosskey.io/v2/

Open banking - PSD2 as a service. (2018, January 17). Crosskey. https://www.crosskey.fi/openbanking/

SPA (single-page application). (2021). https://developer.mozilla.org/en-US/docs/Glossary/SPA

Spring Boot. (2021). https://spring.io/projects/spring-boot

Swagger API documentation. (n.d.). https://swagger.io/resources/articles/documenting-apis-with-swagger/

Swagger UI. (2021). https://swagger.io/tools/swagger-ui/

Techopedia. (2011, July 4). *Front and Back Ends*. Techopedia.com; Techopedia. https://www.techopedia.com/definition/24794/front-and-back-ends

The Official YAML Web Site. (n.d.). https://yaml.org/

Tutorial: Using thymeleaf. (n.d.). https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

van Mulligen, E., & Timmers, T. (1994). Beyond clients and servers. *Proceedings / the ... Annual Symposium on Computer Application [sic] in Medical Care. Symposium on Computer Applications in Medical Care*, 546–550.

Vega, C. (2017, February 28). *Client-side vs. server-side rendering: why it's not all black and white*. freeCodeCamp.org. https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/

WebDriver. (2021). Selenium. https://www.selenium.dev/documentation/webdriver/

What is a REST API? (2020). https://www.redhat.com/en/topics/api/what-is-a-rest-api

What is Java Spring Boot. (2020). https://www.ibm.com/cloud/learn/java-spring-boot

What is REST? (2021). Codecademy. https://www.codecademy.com/article/what-is-rest