



Expertise  
and insight  
for the future

Niklas Kuusisto

# Development of a Multi-Platform Dictionary Application

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

22 May 2019

Author Title	Niklas Kuusisto Development of a Multi-Platform Dictionary Application
Number of Pages Date	38 pages 22 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department Pasi Kråknäs, Head of Training
<p>The goal of this thesis was to develop a multi-platform dictionary application, which works both on modern smartphones and web browsers. The application was developed for NRC Group Finland, which is a company specialised in design, construction and maintenance of railway infrastructure in Finland.</p> <p>The translation application was developed using the Ionic framework and allows users to translate railway-specific vocabulary from one language to another. The dictionary of the application is saved in a SQL database, which is accessed using a RESTful API developed with Python's Flask framework. The vocabulary of the application can be updated with new vocabulary, or existing vocabulary can be changed using a web-based management interface, developed using the Angular framework.</p> <p>The thesis presents the technologies used during the development of the application, such as the Angular, Ionic and Flask frameworks. The thesis also explains the reason these frameworks were used to develop the different parts of the application.</p> <p>The result of the thesis was an application that succeeded in fulfilling most requirements. However, due to time restraints the application could not be developed far enough to fulfill all of them. Development of the application was continued after the completion of this thesis.</p>	
Keywords	Ionic, Angular, Flask, I18n, ADAL JS

Tekijä Otsikko	Niklas Kuusisto Development of a Multi-Platform Dictionary Application
Sivumäärä Aika	38 sivua 22.5.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka (AMK)
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Janne Salonen, Yliopettaja Pasi Kråknäs, Koulutuspäällikkö
<p>Opinnäytetyön tavoitteena oli kehittää monialustainen sanakirjasovellus, joka toimii moderneissa älypuhelimissa, sekä verkkoselaimessa. Työn toimeksiantajana on NRC Group Finland Oy, joka on infrahankkeiden suunnittelutoimisto, rakennusliike ja kunnossapitoyritys.</p> <p>Sanakirjasovellus kehitettiin käyttäen Ionic-kehityskehystä. Käännössovellus mahdollistaa rautatiespesifisen sanaston kääntämisen kieleltä toiselle, esimerkiksi suomesta englanniksi. Sanakirjasovelluksen sanasto on tallennettu SQL-relaatiotietokantaan, jota voidaan hallinnoida Flask-kehityskehyksellä luodulla REST-rajapinnalla. Sanakirjasovelluksen hallintaa varten luotiin Angular-kehityskehyksellä verkkopohjainen hallinnointisivu, jossa sanakirjasovelluksen sanastoa voidaan täydentää, muuttaa tai poistaa sovelluksen valmistumisen jälkeen.</p> <p>Opinnäytetyödokumentti käy läpi opinnäytetyössä käytetyt sovelluskehitysteknologiat, kuten Angular-, Ionic- ja Flask-kehityskehykset. Opinnäytetyödokumentissa selitetään myös, miksi sovelluksen eri osa-alueissa päädyttiin käyttämään tiettyjä sovelluskehityskehyskiä.</p> <p>Opinnäytetyön lopputuloksena syntyi sovellus, joka täytti suurelta osin sille annetut vaatimukset. Ajanpuutteen vuoksi sovellusta ei voitu keittää niin pitkälle, että kaikki sille annetut tavoitteet olisi saavutettu. Sovelluksen kehittämistä jatkettiin opinnäytetyön valmistumisen jälkeen.</p>	
Avainsanat	Ionic, Angular, Flask, I18n, ADAL JS

## Contents

### Glossary

1	Introduction	1
2	Background	1
2.1	Basis for the Application	1
2.2	Requirements	2
2.3	Technology	3
2.3.1	Angular	3
2.3.2	Cordova	5
2.3.3	Ionic Framework	6
2.3.4	Flask	7
2.3.5	MySQL	8
2.4	Theory	8
2.4.1	Model View Controller	8
2.4.2	Single Page Application	9
2.4.3	Hybrid Application Development	10
3	Design and Architecture	11
3.1	Application Architecture	11
3.2	Application design	12
3.3	Software stack	13
4	Application Development	14
4.1	Development environment	14
4.2	NRC Translator	16
4.2.1	Structure	16
4.2.2	Theming Ionic Components	17
4.2.3	Adding Internationalization	18
4.3	Management Interface	21
4.3.1	Creating an Angular Project	21
4.3.2	Structure	22
4.3.3	Setup and Angular Material	22
4.3.4	Angular Router	24

4.3.5	HTTP requests	25
4.3.6	Angular Material Table	25
4.3.7	Angular Material Dialog	28
4.3.8	Adding Authentication	29
4.4	Flask Back End	32
4.4.1	Blueprints and routes	32
4.4.2	Database and models	33
4.4.3	Authentication	34
5	Summary	34
	References	36

## Glossary

AJAX	Asynchronous JavaScript and XML. Technique used to create asynchronous web applications.
API	Application Programming Interface.
CLI	Command-line Interface. A means of interacting with a computer where the user uses text commands to issue orders.
CORS	Cross-Origin Resource Sharing.
CSS	Cascading Style Sheets. Language for describing the presentation of a document written in HTML.
CSV	Comma-separated values. A filetype that uses a comma to separate values.
DBMS	Database management system. Software for maintaining, querying and updating data and metadata in a database.
DI	Dependency Injection. A technique whereby one object supplies the dependencies of another object.
DOM	Document Object Model. API that treats an HTML document as a tree structure.
HTML	Hypertext Markup Language. Markup language used for creating web pages.
HTTP	Hypertext Transfer Protocol.
JSON	JavaScript Object Notation. Human-readable way to transmit JavaScript objects.
ORM	Object-relational mapping. The set of rules for mapping objects in a programming language to records in a relational database, and vice versa.

OS	Operating System. The low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals.
REST	Representational State Transfer.
SPA	Single Page Application.
SQL	Structured Query Language. A language designed for managing data held in a relational database.
UI	User Interface. A visual interface to allow a user to operate software to produce a desired result.
URL	Uniform Resource Locator.

## 1 Introduction

Every industry has industry-specific terminology and jargon, which is used in day to day communication. These consist of words, phrases, and abbreviations, which are used to describe industry tools, tasks, and standards and communicate more conveniently in industry-specific subjects [1]. Vocabulary is usually learned over the years while working in a specific industry from co-workers or during training. This means that new-comers and people not familiar with the industry can have a hard time communicating about industry topics without the knowledge of industry-specific vocabulary. Additionally, the terminology is usually not the same between different languages, which means that translating documents with industry-specific terminology in them can cause problems.

Different industries usually have dictionaries with industry-specific terminology listed in them. However, these dictionaries typically do not include jargon in them and can become outdated as industry standards develop.

The purpose of this thesis was to develop a mobile-friendly dictionary application, called NRC Translator for NRC Group Finland, which would allow employees of the company quickly find the meaning of railway-specific vocabulary and to translate terminology from one language to another. The thesis will address the background, development, and technologies used in the translation application.

## 2 Background

### 2.1 Basis for the Application

During the Metropolia University of Applied Sciences Innovation Project, a proof of concept of the dictionary application was created. The purpose of this proof of concept was to investigate the feasibility and general structure of the dictionary application. The application developed during this thesis is built on the proof of concept.

The Ionic framework was used to develop the front end of the proof of concept application. The proof of concept consisted of 3 views: Home, Item, and Settings. The home view is the main view of the application and is responsible for language choices



and listing search results. In addition, users navigate to other views using the home view. The item view contains the details of each translation, such as the original word in the source language, the translation in the target language, a description, and an example, if available. The settings view allows the user to download a local copy of the dictionary, to enable an offline mode, and to limit the search results to 50 entities.

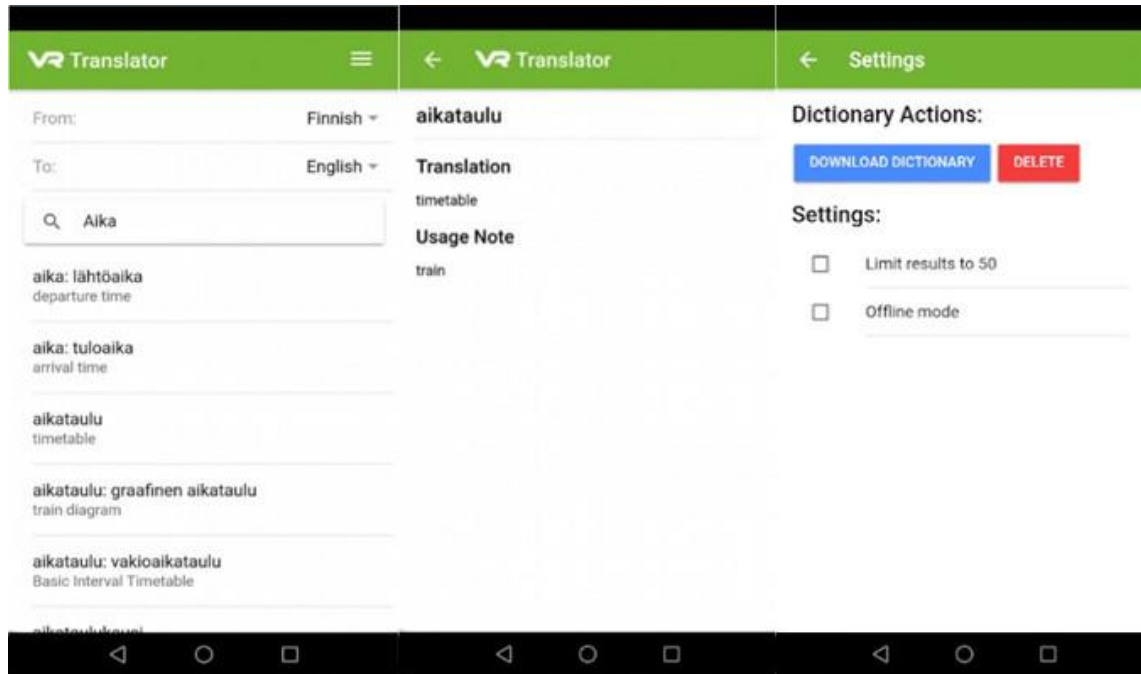


Image 1. Screenshots of the proof of concept Android application

As the proof of concept was built with the Ionic framework, the application allowed for an Android application to be built on the same code base as the web application. Screenshots of the user interface (UI) of the proof of concept application can be seen in Image 1.

## 2.2 Requirements

This thesis aimed to develop an easy-to-use application that is usable on modern smartphones. The UI of the application should be as simple as possible to use and be available in both English and Finnish.

The dictionary of the application should be easy to maintain so that words and languages can be added without additional development. To satisfy this requirement, a separate

administration tool needed to be developed, to manage the application once development was completed. It should be possible to add, view, edit and delete words and languages using the management interface. Additionally, the management interface should accept the upload of new words via CSV files.

Another requirement of the application was to restrict access to the application. Only employees of the company should be allowed access to the application. To achieve this requirement, the identity of users would need to be verified before users are granted access to the application. Access rights to the application would also need to be removed from users who have left the company.

## 2.3 Technology

### 2.3.1 Angular

Angular is an open-source web application framework primarily developed by Google. It was initially released in September of 2016 as Angular 2 but was renamed Angular due to confusion among developers [2]. Angular is one of the most popular front-end frameworks in 2019 and is mentioned in 59% of enterprise full-stack developer job postings. However, recent data suggests that other front-end frameworks like React and Vue are shrinking its market share due to their growing popularity [3]. This change in Angular's popularity is illustrated in Figure 1.

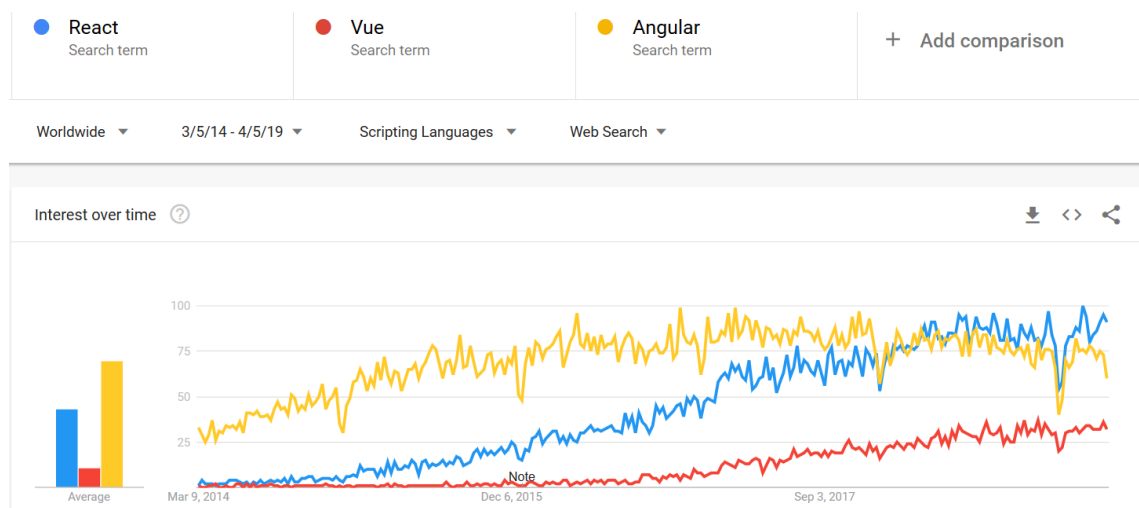


Figure 1. Interest over time in Angular, React and Vue in Google web searches 2014 – 2019 [4]

Angular is written in TypeScript, which is an open-source programming language developed and maintained by Microsoft. Made public in October of 2012, it was designed for the development of large-scale applications. Once compiled, Typescript compiles into JavaScript and the language is a typed superset of JavaScript. TypeScript being a typed subset of JavaScript means that programs written in JavaScript are automatically already TypeScript programs. The advantages TypeScript has over JavaScript has been the addition of a built-in module system, static type checking, classes, interfaces and generics [5].

The basis of Angular applications are `NgModules`. Angular applications are defined by a set of `NgModules` and have at least one `NgModule` as a root module, usually called `AppModule`. The root module is responsible for bootstrapping the application, meaning it loads the rest of the application. `NgModules` consolidate components, directives, and pipes into distinct blocks of functionality, helping keep the project structure lean and concise. In addition, `NgModules` are used to add services to the application and are TypeScript classes marked with the `@NgModule` decorator [6]. An example of an `NgModule` can be seen in Figure 2.

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

Figure 2. Code snippet of Angular CLI created root `NgModule` [7].

The UI of an Angular application is built by using Angular Components. These are TypeScript classes with the `@component` decorator and control a patch of the screen called a *view*. The actual appearance of the view is defined in the template and style

sheet of a component, while the *view* logic, such as data and event binding are defined inside the component class. The template of a component is a form of HTML that is then altered based on the logic and state of the application [8].

While an Angular Component's role is to handle view-related functionality, angular supports creating *Service* classes, to which certain tasks can be delegated, such as fetching data from the backend, logging data and validating user input. This way developers can keep view related tasks separate from data processing, and reuse services in other components should the need arise. Defining a TypeScript class as a *Service* happens with the use of the `@Injectable` decorator.

Dependency Injection or DI happens with the use of the `@Injectable` decorator. The `@Injectable` decorator can be used to register the *Service* at a certain level, such as root. Defining the *Service* as root level makes it available from everywhere in the application. Services can also be provided at specific Modules or Components, by adding the *Service* to the provider's property in a Modules `@NgModule` decorator or a Components `@Component` decorator [9]. An example of an Angular *Service* can be seen in Figure 3.

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor() { }

}
```

Figure 3. Code snippet of an Angular *Service* [10].

### 2.3.2 Cordova

Apache Cordova is an open-source mobile development framework released in 2009. Cordova allows the development of hybrid mobile applications with HTML, CSS, and JavaScript. Cordova works by creating a Native WebView component that displays the

application's HTML, CSS and JavaScript. This way a multiplatform application can be developed with the same code base for each platform. Access to the device's native functions, such as the camera or storage, is done via Cordova Plugins, which are translated to the device's Native OS API calls [11]. A Diagram of the Architecture of a Cordova application can be seen in Figure 4.

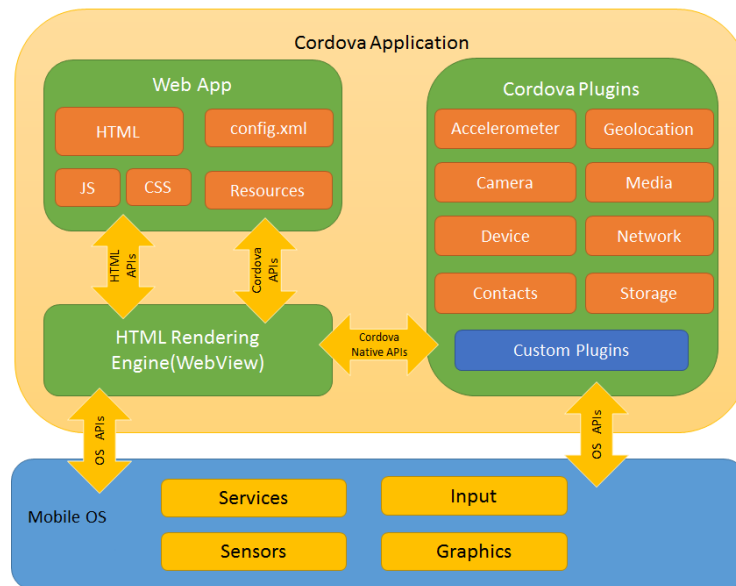


Figure 4. Diagram of the architecture of a Cordova Application [12]

### 2.3.3 Ionic Framework

Ionic Framework is an open source mobile UI toolkit for developing cross-platform applications for iOS, Android, and the web. Ionic provides a library of UI components, with which developers can build hybrid applications with HTML, CSS, and JavaScript. Ionic components use Angular as their base and allow access to native features such as the camera, the GPS, and storage via Cordova plugins. Ionic 4, which came out in January of 2019, adds integration support for Vue and React as frontend frameworks.

Ionic Components are high-level building blocks that developers can use to create mobile-friendly UI elements, such as cards, lists and tabs. Ionic Components usually have HTML elements to define the position in the view and API methods to change the properties of the element. Ionic Components have a prebaked style but can be easily themed with the use of CSS properties. Additionally, Ionic components can change their

look and behavior based on what platform the application is running on (see Image 2) [13].

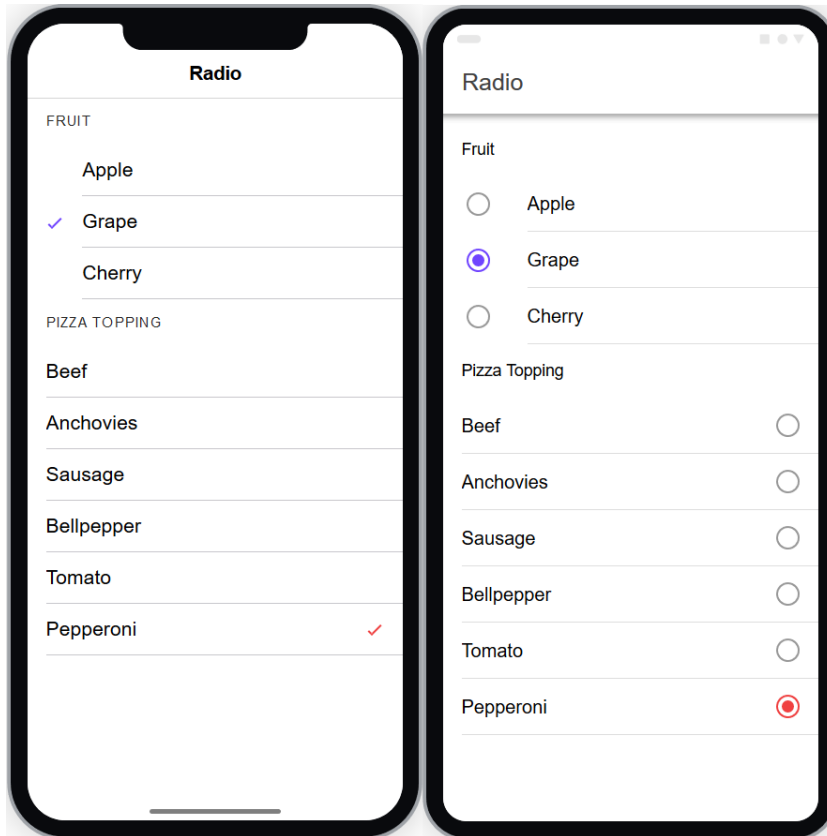


Image 2. Illustration of iOS and Android platform styles differences, while using the same code base in both platforms.

#### 2.3.4 Flask

Flask is a web framework for the Python programming language. It was initially released in 2010 as an April fool's joke. It is designed to be a microframework, meaning that it provides tools to create API routes without the need to import a large number of libraries and dependencies to start development. While the core of the Flask framework lacks features such as database abstraction and form validation, it supports the addition of extensions that add additional functionalities. The result of this design principle is a framework that only requires minimal setup to create a simple web application [14]. An example of a simple Flask application can be seen in Listing 1.

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

Listing 1. Flask web application that prints “Hello World”

### 2.3.5 MySQL

MySQL is an open source relational database management system and is free to use under the GNU General Public License. Developed by Oracle, it is currently the world’s most used database [15,16]. Relational databases such as MySQL store similar data into tables. Tables contain individual entries, that are called rows. Each row has fields, which contain the data about the record, such as the ID and Name. Each individual row must contain a unique identifier also called a primary key, which is used to identify each individual record [17]. Fields can also contain a reference to the primary key of a row in another table. These references are called foreign keys.

## 2.4 Theory

### 2.4.1 Model View Controller

The Model View Controller (MVC) pattern is an architectural pattern that aims to separate the UI (View) from the underlying application logic (Controller) and application data (Model). The advantage of using the MVC pattern is that software components that abide by its standards are modular and more easily reusable [18]. Components that are separate from each other are easier to replace or change, without the need to do significant changes to the other components of the application. A diagram of the different MVC components can be seen in Figure 5.

The views of the application, which are the part that the user sees and interacts with, should only contain logic for passing the users commands to the controller and updating itself based on the data received from the controller, or based on the changes in the model.

Controllers should be restricted to accepting input from either the view or the model, converting it to commands readable by the rest of the application, and passing it on to either the view or model. An example of this would be to accept new data from the view, validating that the new data is correct and passing commands to the model to update itself according to the received data.

The model of the application is responsible for the data. Such as the data's structure, making changes to it and how it is saved/retrieved. Depending on the implementation it might also validate the instructions passed to it from the controller. The model of an application might be the classes that define the different types of products of an online shop and an SQL Database where data on the products is stored [19].

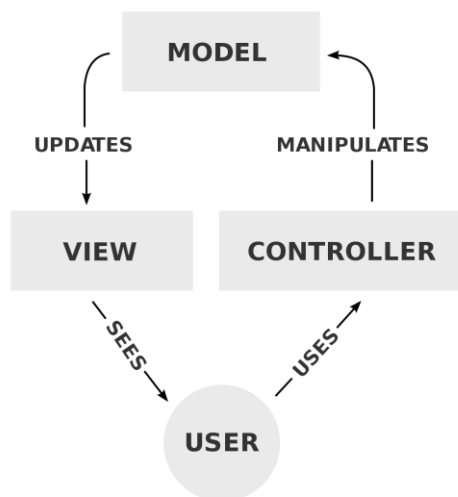


Figure 5. Diagram of interactions between the different MVC components and the user [20]

#### 2.4.2 Single Page Application

Single Page Applications (SPA) are web applications that load in a single HTML page, which is then updated according to the user's actions with the use of AJAX calls. Traditional web applications render a new web page every time a user sends a request to the server. In single page applications, AJAX requests return data to the client, which is then added to the client page using DOM manipulation (see Figure 6.).



Single page applications are more fluid and responsive as the browser does not need to render an entirely new web page on each page reload. Single page applications also separate view logic from the application logic making it easier to adhere to the MVC pattern, gaining it the advantages that adhering to it brings [21].

The disadvantage of single page applications is that the loading of the initial application takes longer due to the increased size of the web page. Single page applications are more vulnerable to cross site scripting (XSS) attacks. The user's browser must also support the use of JavaScript for SPA's to work, as JavaScript is used to edit the content of the page [22].

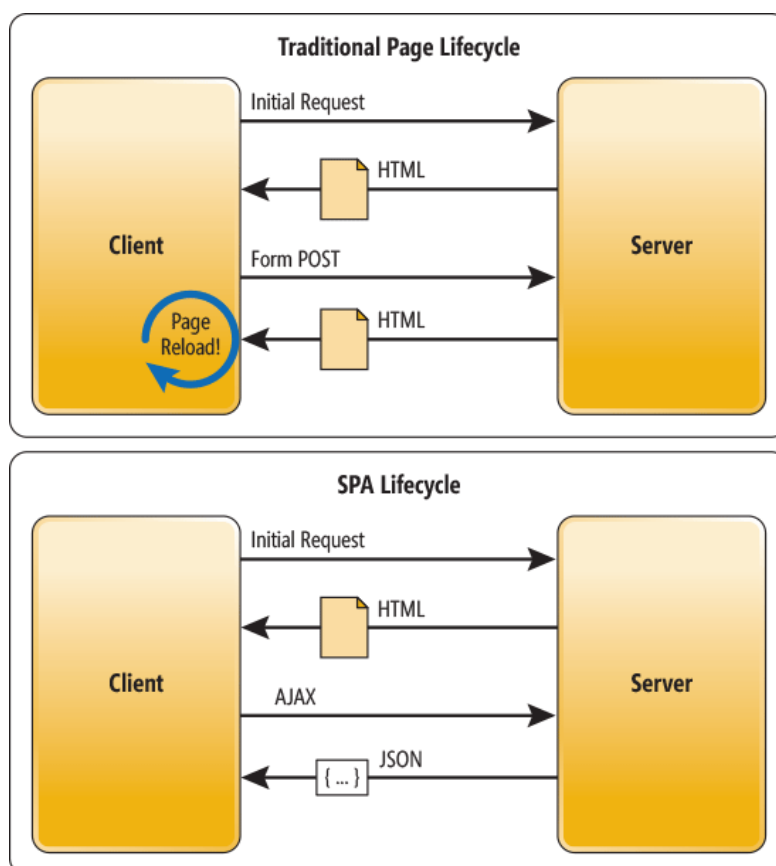


Figure 6. Diagram comparing the lifecycle of a traditional web application to an SPA [21].

### 2.4.3 Hybrid Application Development

Hybrid applications are applications that are developed using with programming languages, such as HTML, CSS, and JavaScript, but can be deployed to run on multiple

devices, such as Android, IOS and Windows Phone. This way an application can be developed for various native devices with less development time and less development expertise needed.

Hybrid development also has some disadvantages compared to developing strictly to a specific platform. While a hybrid application can be developed for different platforms such as Android and iOS with the same codebase, the user interface guidelines for different platforms might be different. This means that an interface developed for a certain platform such as Android cannot be directly ported to another platform such as iOS, due to the platform specific guidelines. UI components that work on a certain platform might also be unavailable or work differently on other platforms. Applications developed purely with a single platform in mind also have better performance [23].

### **3 Design and Architecture**

#### **3.1 Application Architecture**

The architecture of the translation application was designed to fulfill the requirements as specified in chapter 2.2. The entire application consists of three distinct components: The translation app, the management interface, and the back end. The function of each component is described below. A diagram of the application architecture can be seen in Figure 7.

The translation app is user-facing part of the application. Its purpose is to allow users to search for translations for certain industry-specific words on their mobile devices and desktop machines. This is done via the native Android application or the Web-based app.

The management interface's purpose is to allow create, read, update and delete (CRUD) operations on the dictionary of the application. Administrators of the application can use this interface to view, add, edit and delete individual words and languages of the application. This component was created, to keep the management of the dictionary application separate from the translation app.

The back end of the application consists of a RESTful API and a SQL database. The database contains the languages and each individual translation needed in the use of the application. Access to the database happens through the API by using the provided URI routes. Both the translation application and the management interface call the back end using HTTP requests to read and write data to the database.

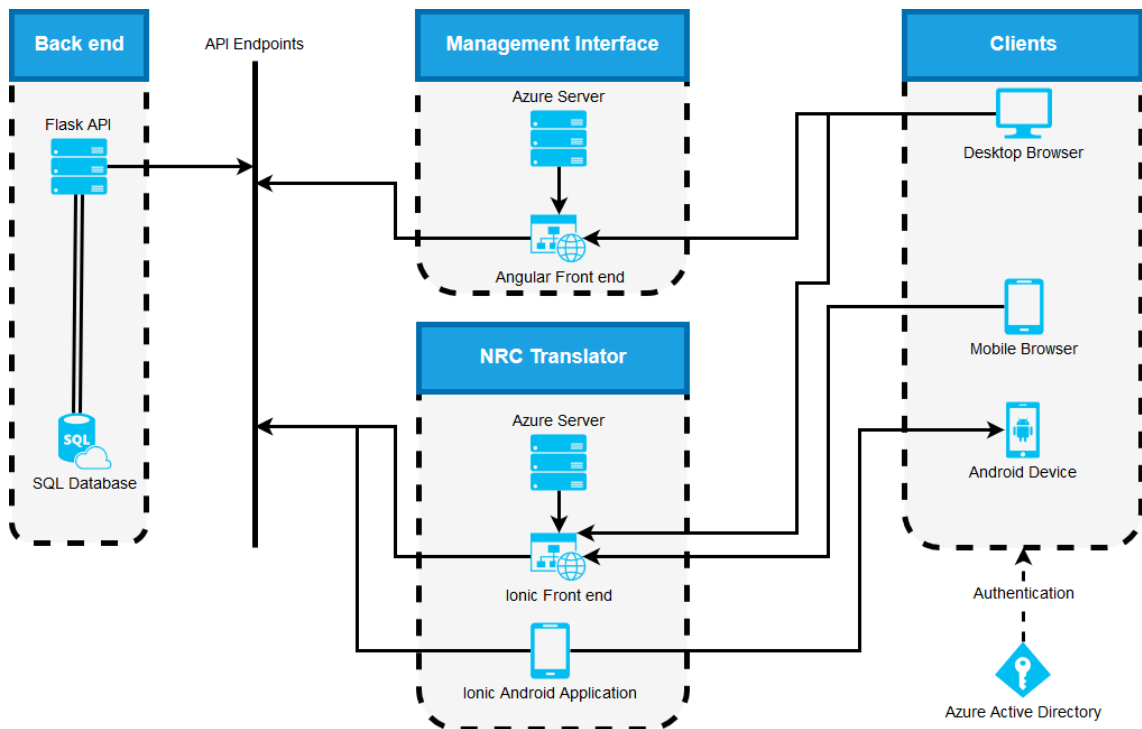


Figure 7. Diagram of the architecture of the application

### 3.2 Application design

As a proof of concept of the Android application was developed before the start of the thesis, much of the initial UI design was done during the development of the proof of concept. Further changes to the UI of the translation application were prototyped in a new branch of the project's git repository before they were shown to the project manager for analysis and feedback.

Design into the management interface, such as changes to the UI and new features, were also discussed on the weekly design meetings. During the design meetings, suggestions about new application features, such as the ability to import new translations

via CSV were proposed. The project manager was informed if it was possible to implement this feature, the approximate time it would take to implement it, and how the users would access the feature in the application UI.

A considerable amount of time was put into designing the user authentication of the application. During the design meetings, two different suggestions for solutions came up. One was to handle authentication with an SSO solution, like Azure AD, which would also be used in other applications of NRC Group Finland. Another suggestion was to create user accounts into the database using the company email address as the username and emailing a password to the email address to make sure that only company employees have access to the application.

It was decided at the translation application would be available to use by everyone in the company. However, access to management interface might only be available to certain personnel in the company, such as the IT staff or management. To achieve this, it was necessary to integrate the applications into the companies Azure Active Directory (Azure AD) as this makes it simple to add/remove user access to the different enterprise applications using the Azure AD portal.

Following the decision to choose Azure AD, time was put into getting familiar with the Microsoft identity platform documentation. The implementation of the different authentication protocols was tested by creating small Angular and flask projects to help choose the correct protocol and authentication approach.

### 3.3 Software stack

Ionic was chosen as the framework for the translation application, as the requirements specified that the application was to run on both web browsers and natively on Android devices. This requirement limited the choice of framework to hybrid application frameworks. The reason to use Ionic over other frameworks was influenced by its open source license, the support to build the application into a website as well as a native application, and the amount of documentation available online.

The choice of framework for the management application was more difficult to make, as any front-end framework could potentially be suitable for the task. Ionic could have been used to develop the application. This way, both front-facing applications could have been developed with the same framework. However, as Ionic is meant primarily for applications that work on mobile devices, its default UI components were not as suitable for desktop environments as other frameworks. In the end Angular was chosen as the framework. Ionic uses Angular as its base, so using Angular would keep the application structure of both frameworks fairly similar to each other, while being more suitable for designing desktop friendly UI.

The back end of the application was written in Python, using the Flask framework. As the back end of the application is primarily only responsible for interacting with the database, and the required computing is light, the primary criteria in choosing a language and framework was easiness to set up an application as a developer. Another important criterion was the support for ORM or Object Relational Mapping. The advantages Flask had over other potential choices were, for example, its excellent third-party library support (such as SQLAlchemy, for ORM), the ability to set up a simple web server in a few minutes as described in listing 1, Python's popularity as a programming language, and focus on readability.

## **4 Application Development**

### **4.1 Development environment**

The application was developed on the Windows operating system using Windows 10 and Microsoft's Visual Studio Code was used to edit the source code of the application. Visual Studio Code is an open source source-code editor developed by Microsoft and is available for Windows, Linux, and macOS. It is not an IDE and differs from Microsoft's Visual Studio by not containing some IDE features, such as Integrated compilers/interpreters and autocompletion [24]. It is, however, very lightweight and only requires around 200 MB of hard drive space [25]. Additionally, there are a large number of extensions available for Visual Studio Code, which allow users to add languages, debuggers, and tools to support development workflow. During the last few years, Visual

Studio Code has become the most popular development environment used by developers, according to the Stack Overflow 2019 Developers Survey [26].

Android Studio's Android SDK was used to test the native Android application, by using its built-in Android emulator. The application was also tested on the developers own Honor 8 smartphone and a Samsung Galaxy A6 smartphone.

Version control was achieved by using git, and the remote was hosted on GitLab. Using git allowed changes to the application to be developed on feature branches, instead of working directly on the master branch of the application. The advantage of this is that the master branch only contains code that has been tested to work. Changes can also be reverted easier, as merges can be reverted to a previous commit if problems arise with a new feature.

## 4.2 NRC Translator

### 4.2.1 Structure

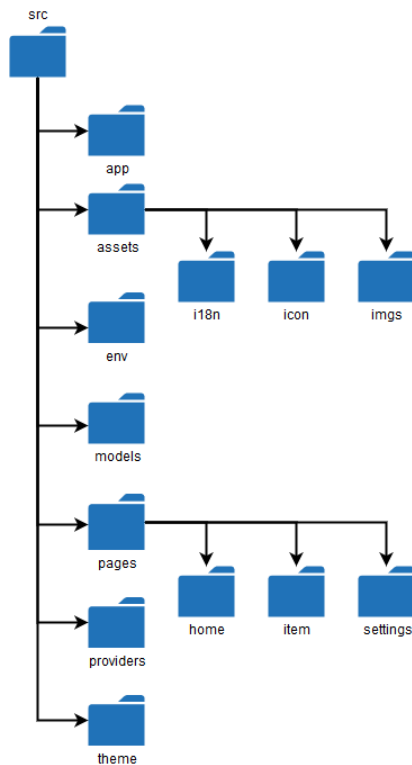


Image 3. Folder Structure of the translation application

The uncompiled source code of an Ionic project is located in the src folder. The structure inside the `/src` folder is similar to a traditional Angular application with the most important folders listed below.

- App: Contains the entry point and files to bootstrap the application.
- Assets: Contains all images, logos and the translation json data required for ngx-translate.
- Models: Contains the Typescript data model used in the application.
- Pages: Contains the Angular components, templates and styles of individual pages of the application.

- Theme: Contains the global Sass themes of the application.

#### 4.2.2 Theming Ionic Components

Ionic components use CSS variables to define their default appearance. This means that Ionic components such as `<ion-toolbar>` can only have some of its properties such as `background-color` changed by overriding its global CSS variable either globally or locally in the template's styles sheet, or by adding a `color` property to the HTML element of the Component.

Much of the appearance of the translation application had to be changed to fit the aesthetics of other NRC Groups applications. Global colour variables were created in the `$colors` map (see Listing 2.) located in `src/theme/variables.css` file which then could be applied to any component of the application.

```
$colors: (
  primary:    #488aff,
  secondary:  #32db64,
  danger:     #f53d3d,
  light:      #f4f4f4,
  dark:       #222,
  twitter: (
    base: #55acee,
    contrast: #ffffff
  )
);
```

Listing 2. Example of Ionic a global colors map [27].

To apply a style to all Ionic components of an application, one must override the default Sass Variable in the projects `src/theme/variables.scss` file. An example of this can be seen in Listing 3 below.

```
$toolbar-background: color($colors, twitter, base);
```

Listing 3. CSS variable that defines the background color of all `<ion-toolbar>` components as the twitter variable defined in the `$colors` map.



### 4.2.3 Adding Internationalization

The requirements specified that the UI of the translation application would need to be in two different languages: Finnish and English. Users should be able to switch the language displayed on the screen with the click of a button. Internationalization or i18n can be added to Ionic projects with the `ngx-translate` library for Angular. This library can easily be installed with the node package manager (npm). As it is recommended to store translation strings in separate files, to limit the amount of hard-coded strings in an application, `ngx-translate` needs a loader to load the translation files. There are multiple available loaders to choose from, but the application developed during this thesis used the `http-loader`, which uses HTTP calls to load the files. The CLI command used to install `ngx-translate` and `http-loader` can be seen in Listing 4.

```
npm install @ngx-translate/core @ngx-translate/http-loader --save
```

Listing 4. Installing `ngx-translate` and `http-loader` using npm

The Installed libraries must be imported into the root module of an application, by importing their modules into the root module file, usually named: `app.module.ts` and adding the `TranslateModule` to the `imports` property of the `NgModule`. Importing the `TranslateModule` requires a definition of which loader to use and what dependencies the loader requires. For this an exported function must be created, to act as the factory, which creates the loader initially. Creation of the `TranslateHttpLoader` requires a reference to the `HttpClient` used by the loader and the prefix and suffix all translation files use. An example of the import can be seen in Listing 5.

```
import { TranslateModule, TranslateLoader } from '@ngx-translate/core';
import { HttpClientModule, HttpClient } from '@angular/common/http';
import { TranslateHttpLoader } from '@ngx-translate/http-loader';

export function createTranslateLoader(http: HttpClient) {
  return new TranslateHttpLoader(http, './assets/i18n/', '.json');
}

@NgModule({
  declarations: [
    /* list of directives and pipes used in the application */
  ],
  imports: [
    /* Other imports of the application */
    TranslateModule.forRoot({
      loader: {
        provide: TranslateLoader,
        useFactory: (createTranslateLoader),

```

```

        deps: [HttpClient]
      }
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})

```

Listing 5. Example of code required to import `ngx-translate` and `http-loader` into the root module of the application.

For `ngx-translate` to work in an application, one needs to create the translation files in the folder defined in the `TranslateHttpLoader` prefix argument. In the case of the example, files must be created in the `src/assets/i18n` folder. The files must also be JSON files as defined in the suffix argument of `TranslateHttpLoader`. An example of the structure of these files can be found in Listing 6. below.

```

{
  "home": "Home",
  "about": "About",
  "contact": "Contact",
  "welcome": "Welcome to This Application Example",
}

```

Listing 6. Example of the structure of a .json translation file used by `ngx-translate`.

To define what translation file `ngx-translate` will use, `TranslationService` provides two different methods: `setDefaultLang(lang: string)`, which defines the language to use if a translation does not exist, or if none have been set, and `use(lang: string)`, which changes the `TranslationService` to use the language provided in the method argument. The language string used in the argument must be the same as the file name of the JSON translation file. An example of this definition can be seen in the Listing 7.

```

translate.setDefaultLang('en');

```

Listing 7. Setting the `TranslationService` to use `en.json` as the default Language.

The Translation pipe can then be used to display the correct text in the components HTML template (Listing 8).

```

<h2>{{"home" | translate }}</h2>
<p>
  {{ "about" | translate }}

```

```

</p>
<p>
  {{ "welcome" | translate }}
</p>

```

Listing 8. Using the translate pipe to display text in HTML template pages

Alternatively, `TranslationService` also provides a `get` method to fetch a translation programmatically in the TypeScript files of the application (Listing 9).

```

translateService.get('contact').subscribe(
  value => {
    // value is our translated string
    let contact = value;
  }
)

```

Listing 9. Using `TranslationService` to get a translation string

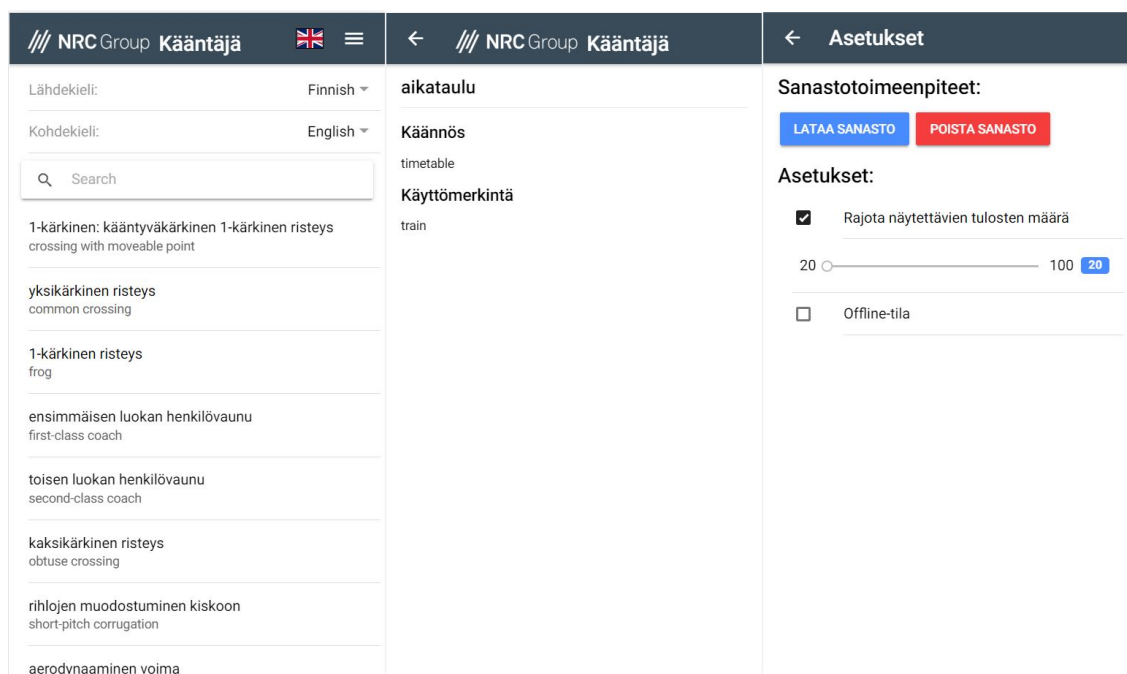


Image 4. Screenshots of the translation application once it had been rethemed and internationalization had been added.

## 4.3 Management Interface

### 4.3.1 Creating an Angular Project

The easiest way to create an Angular project is to use the Angular CLI (Command line interface), this can be downloaded using the npm, with the following terminal/console command:

```
npm install -g @angular/cli
```

after installing the CLI, creating an Angular project is simple. An Angular workspace project, with a specified project name, and an Initial skeleton app can be created with the command:

```
Ng new example-app
```

The command will also prompt the caller with features that can be included in the application, such as Angular Routing and which stylesheet format to use. This will create an Angular workspace in a folder named `example-app` in the current working directory. The folder will include an initialized local git repo, a sample application inside the `src` folder, an end-to-end text project in the `e2e` folder, and configuration files.

### 4.3.2 Structure

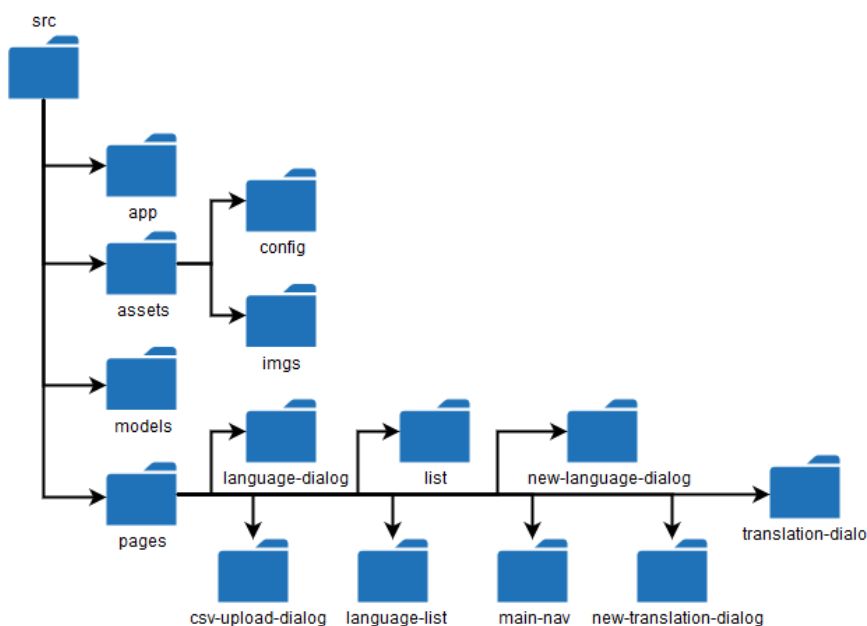


Image 5. Folder structure of the management interface

As Angular components are used in Ionic, the folder structure is similar to the translation application. As with the Ionic project, the uncompiled source code is located in the `src` folder inside the workspace of the application. The only major change in the structure of the management interface is that removal of the `env`, `providers` and `theme` folders.

### 4.3.3 Setup and Angular Material

Development on the UI of the application started by generating a new project via the Angular CLI, as per chapter 4.3.1. The Included example applications source code was removed, and work was begun on the main navigation view.

The main navigation view's template consists of the main header of the application, and a sidebar to switch between the two different pages of the application. As the management application is an SPA, the actual HTML page would not change, but the page contains a router-outlet, the content of which would vary depending on the view and URL. To speed up development a UI component library called Angular Material was installed using `npm`.

```
npm install --save @angular/material @angular/cdk @angular/animations
```

For Angular Material components to render correctly in an application, one must import the `BrowserAnimationsModule` to enable animations, or alternatively `NoopAnimationsModule` to disable them. Additionally, installing an Angular Material core theme is recommended as Angular Material components will not work without installing a theme. Angular includes several prepackaged themes. A theme can be included by entering the following line in `styles.css` to import it globally, or in a component's style sheet, to apply the styles to only that components template.

```
@import '@angular/material/prebuilt-themes/indigo-pink.css';
```

Pre-Built Angular Material themes:

- `deeppurple-amber.css`
- `indigo-pink.css`
- `pink-bluegrey.css`
- `purple-green.css`

The Angular Material library provides responsive ready-made UI components that speed up the development of user interfaces and can be styled to fit an application's theme if needed. Angular Material also allows developers to generate UI component blueprints, using the schematics included in the `angular-material` package. A navigation component can be generated with the following CLI command:

```
ng generate @angular/material:nav <component-name>
```

This command will generate a folder with the component name in `src/app` containing a ready-made `toolbar` and `sidenav` components. This component can then be rendered by calling its CSS selector in the template of an Angular component (see Listing 10).

```
<app-main-nav></app-main-nav>
```

Listing 10. Calling the CSS selector of the generated navigation component

### 4.3.4 Angular Router

To change the view of a page if a user clicks a link or when the correct URL is entered in the address bar of the browser the Angular `Router` is used. The Angular Router is an optional service that maps URL paths to different components. The user can then navigate to these by following a link to the URL or typing the URL directly in the browser's address bar. To use the router service in an application, it must be configured first by defining the application's routes (see Listing 11). These consist of an array of routes that are passed to the `RouterModule.forRoot()` method in the imports of an Angular module. A route consists of a URL path and a component. The path is written relative to the URL of the application and can include parameters in it, the value of which can be retrieved in the component the route is mapped to. URL paths An URL path can also be defined to redirect to another URL path.

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
  { path: '', redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

Listing 11. Example route definition [28].

The `RouterOutlet` directive is used to define where a component renders when routed to. By placing the directive on the host component's template the router will render components as a sibling element to the `RouterOutlet`. To allow users to navigate to the component without typing the route's URL in the address bar the `RouterLink` directive is used (see Listing 12). Clicking on a `RouterLink` will change the URL of the browser to the path assigned to the `RouterLink` directive [29].

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

Listing 12. Angular template containing the `router-outlet` and `routerLink` directives [30].

### 4.3.5 HTTP requests

In order to communicate with backend services, HTTP requests are required to fetch data from the API. Angular provides a simplified HTTP API with the `HttpClient` module, which uses the browser's `XMLHttpRequest` API to send HTTP requests. Like other Angular components, this module must first be imported in the root `NgModule` of the application and the component or service it is used in. Once imported, the `HttpClient` provides multiple methods which can be called to create appropriate HTTP requests, such as `get`, `post`, `put` and `delete`. Because receiving a reply to an HTTP request is time-consuming, the methods return an `Observable`, which can be subscribed to, making the methods asynchronous. An example of the method can be seen in Listing 13.

```
this.http.get(url).subscribe(data => {  
  console.log(data);  
}, error => {  
  Console.log(error);  
});
```

Listing 13. Using Angular to send a get request and logging the result in the console of the browser.

### 4.3.6 Angular Material Table

As the primary function of the management interface is to view and edit the data of the translation application, it needs to display rows of data (See Image 6). Angular Material provides a styled data-table, which can be easily be expanded upon as developers have full control over the interaction patterns of the table. An Angular Material data table is added to a template with the `<table mat-table>` component (see listing 14). To define the data of the table, a `dataSource` input is provided by the data table component, to which an array containing objects can be passed. The `dataSource` input also accepts `DataSource` classes, such as `MatTableDataSource`, which provides a way to encapsulate any sorting, filtering, pagination, and data retrieval logic of the application.

```
<table mat-table [dataSource]="myDataArray">  
  ...  
</table>
```



Listing 14. A variable called `mydataArray` is passed as the `dataSource` to an Angular Material data table

Each column in the data table needs to have a column definition. The column definition defines the title of the header and the content of the cell. It is specified via a `<ng-container>` with the `matColumnDef` directive, which is used to assign the column definition a unique name. A column's title is defined via a `<th>` tag with a `mat-header-cell` attribute and `*matHeaderCellDef` directive as its attributes, with the content between that start and end tag defining the displayed cell header. A column's content is defined with a `<td>` tag that has a `mat-cell` attribute and `*matCellDef` input directive as its attributes, with the content between the start and the end `<td>` tag containing the displayed content. Additionally, the footer of a cell can be defined with the content between a `<td>` tag with a `mat-header-cell` attribute and `*matHeaderCellDef` directive. An example of a column definition can be seen in Listing 15.

```
<ng-container matColumnDef="id">
  <th mat-header-cell *matHeaderCellDef> ID </th>
  <td mat-cell *matCellDef="let tableData"> {{tableData.id}} </td>
</ng-container>
```

Listing 15. Example of a column definition

Once the columns of the table have defined, the data table the rows of the table need to be defined. The header row is defined with a `<tr>` tag containing a `mat-header-row` attribute and a `*matHeaderRowDef` directive. A `<tr>` tag also defines the template for an individual row. However, this tag needs to contain a `mat-row` attribute and `*matRowDef` directive as its attributes. The row definitions also dictate what columns are shown on the table, this is done by defining a variable in the host component of the table with an array containing the names of the columns to be rendered on the table (see Listing 16). This array is passed to the header row definitions `*matHeaderRowDef` directive and the row templates `*matRowDef` directive (See Listing 17).

```
columnsInTable = ['id', 'data'];
```

Listing 16. Array containing the names of columns to render in a data table

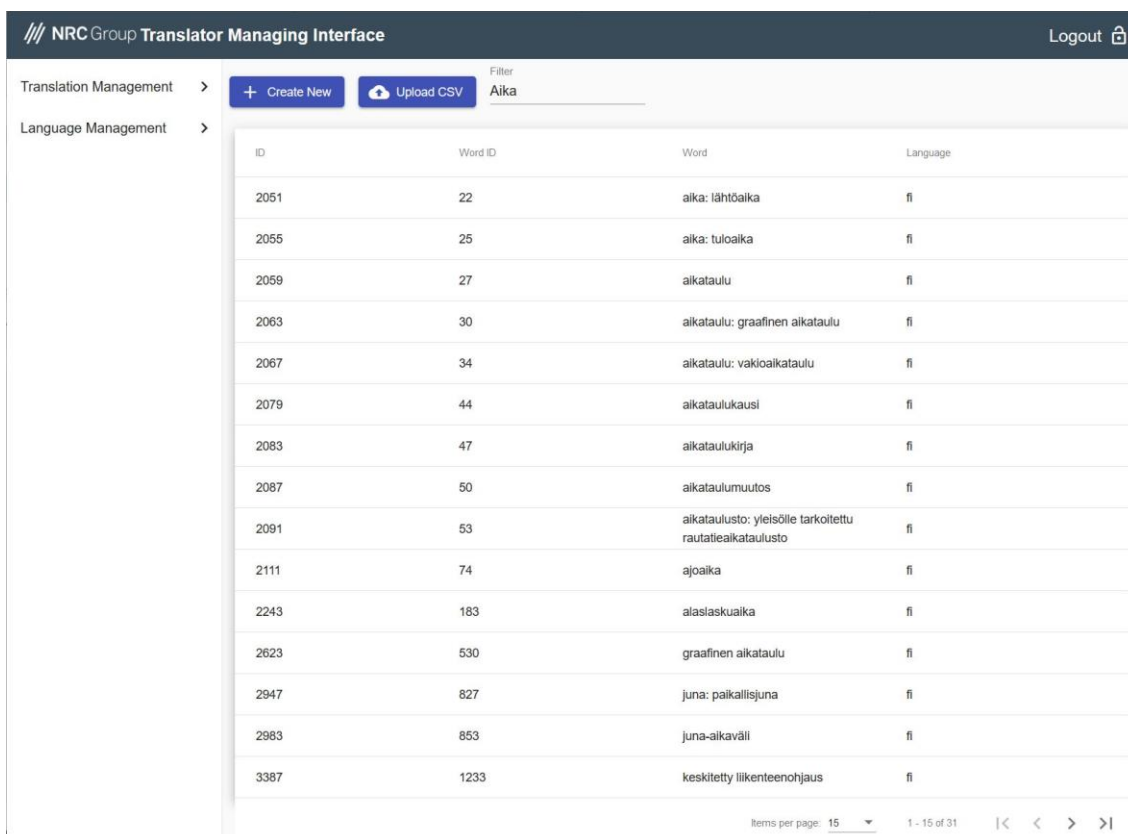
```
<tr mat-header-row *matHeaderRowDef="columnsInTable"></tr>
<tr mat-row *matRowDef="let row; columns: columnsInTable"></tr>
```

Listing 17. Example of row definitions

When listing large amounts of data on a table, adding pagination, sorting and filtering the data table is suggested, as it makes it easier for a user to find data. As mentioned earlier it is recommended to use `MatTableDataSource` for the data source of the table, as using it as the data source will not require filtering, pagination and sorting logic to be set up [30]. An example of a `MatTableDataSource` can be seen in Listing 18.

```
mydataArray = new MatTableDataSource([
  { id: 1, data: 'row1 data'},
  { id: 2, data: 'row2 data'},
  { id: 3, data: 'row3 data'},
  { id: 4, data: 'row4 data'},
  { id: 5, data: 'row5 data'},
  { id: 6, data: 'row6 data'},
]);
```

Listing 18. Setting up the data source of a table as a `MatTableDataSource` with initial data



The screenshot shows the 'NRC Group Translator Managing Interface' with a 'Logout' button in the top right. The main content area is titled 'Translation Management' and includes a '+ Create New' button, an 'Upload CSV' button, and a search filter labeled 'Aika'. Below this is a table with the following data:

ID	Word ID	Word	Language
2051	22	aika: lähtöaika	fi
2055	25	aika: tuloaika	fi
2059	27	aikataulu	fi
2063	30	aikataulu: graafinen aikataulu	fi
2067	34	aikataulu: vakioaikataulu	fi
2079	44	aikataulukausi	fi
2083	47	aikataulukirja	fi
2087	50	aikataulumuutos	fi
2091	53	aikataulusto: yleisölle tarkoitettu rautatieaikataulusto	fi
2111	74	ajoaika	fi
2243	183	alasiasku aika	fi
2623	530	graafinen aikataulu	fi
2947	827	juna: paikallisjuna	fi
2983	853	juna-aikaväli	fi
3387	1233	keskitetty liikenteenohjaus	fi

At the bottom of the table, there is a pagination control showing 'Items per page: 15' and '1 - 15 of 31'.

Image 6. Screenshot of the management interface. A `<table mat-table>` is used to list translations fetched from the API.

### 4.3.7 Angular Material Dialog

Displaying all fields of an entry shown in the data table can be unfeasible, due to the lack of available screen space, especially when using a mobile device, such as a mobile phone. Viewing an individual entry and its fields is better suited to a popup modal that can be dismissed easily (See Image 7). Angular Material contains a dialog service that can be used to create a modal dialog, which includes this functionality. A dialog can be created and rendered on the screen, by calling the `open()` method of `MatDialog` service. The `open()` method requires a reference to a component, which will be rendered as the content of the dialog, and `MatDialogConfig` object, which contains the configuration for the dialog, such as the height, width and the data passed to the dialog component. the method will also return an instance of `MatDialogRef`, which is a handle of the opened dialog. It can be used to perform actions on the opened dialog, such as updating its size and closing it. An example of a `MatDialog` definition can be seen in Listing 19.

```
let dialogRef = dialog.open(DialogComponent, {
  height: '400px',
  width: '600px',
  data: {name: 'Niklas', age: '24'},
});
```

Listing 19. Creation of a dialog

Data passed to the dialog using the `data` option as seen in listing 19 can be accessed in the dialog component with the `MAT_DIALOG_DATA` injection token. Data can also be passed back from the dialog when injecting the `MatDialogRef` to the dialog component and closing it in the component with the `close()` method. An example of a dialog component can be seen in Listing 20.

```
import {Component, Inject} from '@angular/core';
import {MAT_DIALOG_DATA} from '@angular/material';

@Component({
  selector: 'your-dialog',
  template: 'passed in {{ data.name }}',
})
export class DialogComponent {
  constructor(@Inject(MAT_DIALOG_DATA) public data: any,
    public dialogRef: MatDialogRef<YourDialog>) { }

  closeDialog() {
    this.dialogRef.close('Pizza!');
  }
}
```

```

}
}

```

Listing 20. Dialog component that displays the name property of the data passed to it. The result 'Pizza!' is passed back from the dialog as the dialog is closed.

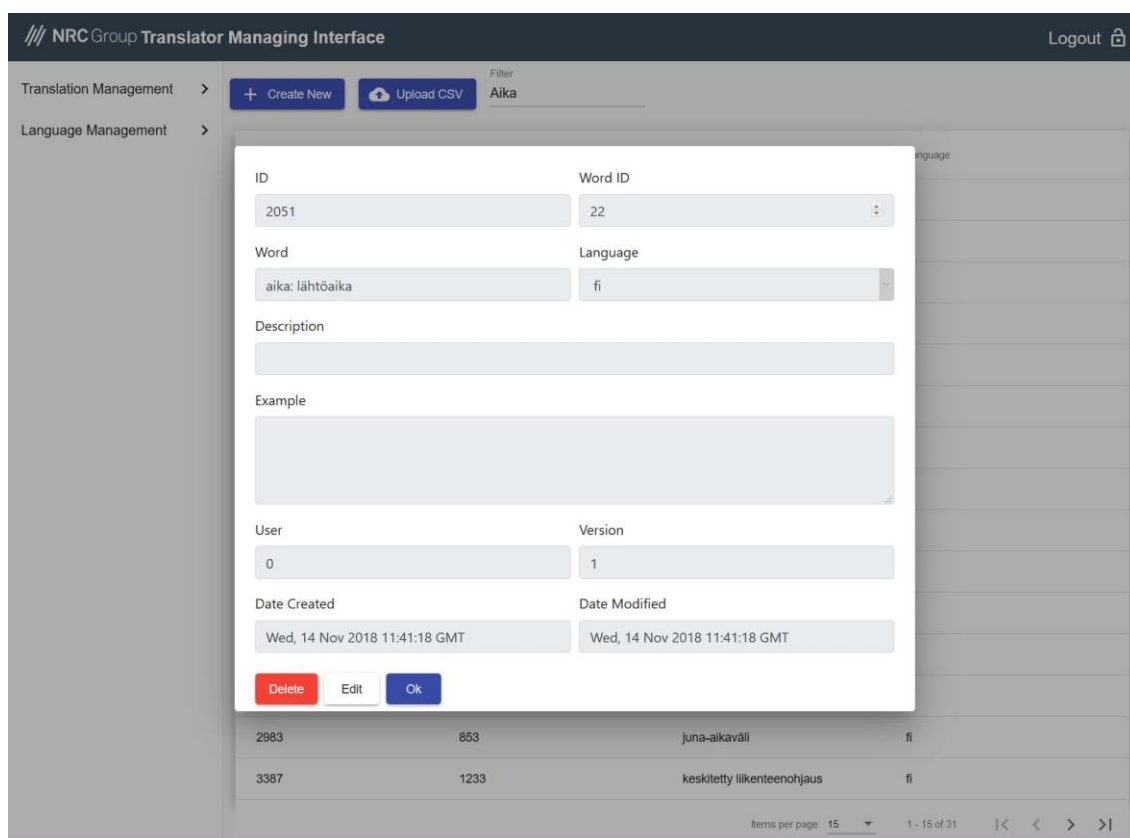


Image 7. Screenshot of a dialog used to edit the details of an individual translation entity.

#### 4.3.8 Adding Authentication

During the design phase the decision was made to use Azure AD as the authorization provider of the application. Azure AD has support for industry standard protocols, such as OAuth 2.0, OpenId Connect and SAML 2.0[32]. The OAuth2 implicit grant flow was used in the translation management interface, as this grant is the grant most suited for SPA applications that consume API's via JavaScript, such as the management interface developed during this thesis [33]. A diagram of this flow can be seen in Figure 8.

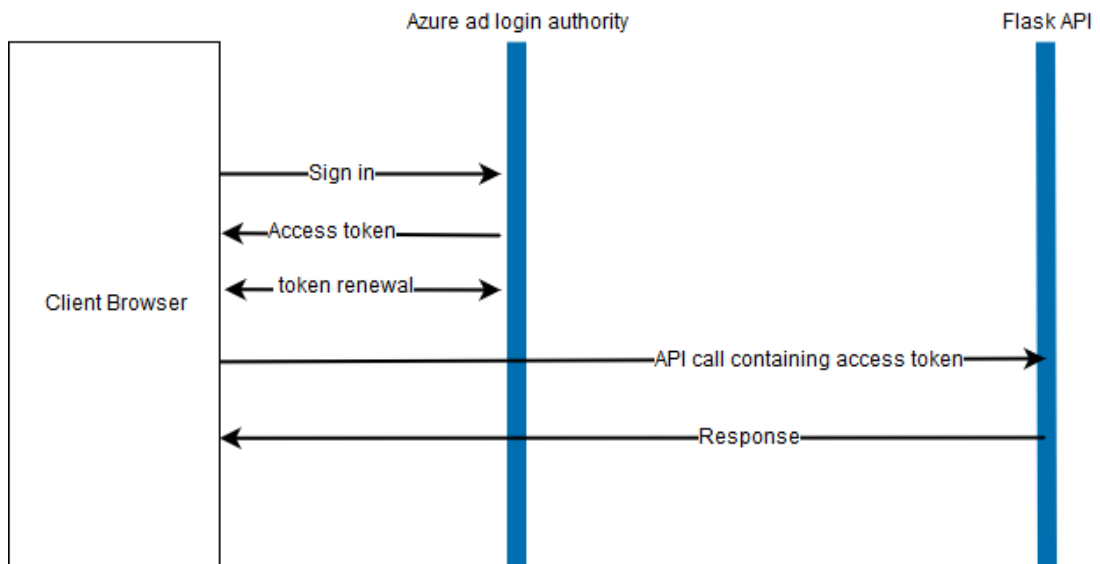


Figure 8. Diagram of the authorization flow

The translation management application uses Active Directory Authentication Library for JavaScript (ADAL JS) to secure the front end of the application. To be able to use ADAL JS in an application a client must first be registered on the Azure portal with `oauth2AllowImplicitFlow` property set to `true` in the application manifest of the registered application. ADAL JS has a wrapper library for Angular, which can be installed via npm with the following code:

```
npm i microsoft-adal-angular6
```

ADAL needs to be configured in the import property of the root module of the application by providing it the tenant id, the client id, cache location, redirect URI and possible API endpoints (see Listing 21).

```
imports: [
  ...
  MsAdalAngular6Module.forRoot({
    tenant: '[INSERT TENTANT ID HERE]',
    clientId: '[INSERT CLIENT ID HERE]',
    redirectUri: window.location.origin,
    endpoints: {
      "[INSERT ENDPOINT HERE]": "[INSER RESOURCE ID HERE]",
    },
    navigateToLoginRequestUrl: false,
    cacheLocation: 'localStorage',
  }),
  ...
]
```

Listing 21. Example of configuring ADAL while importing the module

Router guards are used in securing the different routes of an Angular application, of which the `CanActivate` interface handles who can navigate to a specific route. The `microsoft-adal-angular6` library already contains a service called `AuthenticationGuard`, which checks whether a user is currently authenticated with AAD to view a route secured with it. Securing routes with `AuthenticationGuard` is simple, the service must be imported and added as a provider in the root module of the application, after which route definitions can be updated with a `canActivate` guard property, to secure them (See Listing 22).

```
const routes: Routes = [
  { path: '', component: AppComponent, pathMatch: 'full',
    canActivate: [AuthenticationGuard]}
];
```

Listing 22. Angular route secured with the `AuthenticationGuard` route guard

As the API of the application must be secured from unauthorized access, all API calls must include an id token. This token is received by ADAL on login and is renewed automatically by ADAL when it expires. Authentication information is usually added as a JSON Web Token as an `Authorization` header with the `Bearer` scheme. As of Angular 4.3, Angular includes the `HttpInterceptor` interface, which can be used to intercept and modify HTTP requests globally. A `HttpInterceptor` is used in the management application to add an `Authorization` header to all HTTP requests that are sent to the endpoints defined in the ADAL configuration. An example of the intercept method can be seen in Listing 23.

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  // get api url from adal config
  const resource = this.adal.GetResourceForEndpoint(req.url);

  if (!resource || !this.adal.isAuthenticated) {
    return next.handle(req);
  }

  // merge the bearer token into the existing headers
  return this.adal.acquireToken(resource).pipe(
    mergeMap((token: string) => {
      const authorizedRequest = req.clone({
        headers: req.headers.set('Authorization', 'Bearer ' + token),
      });
      return next.handle(authorizedRequest);
    });
  );
}
```

```

    }));
}

```

Listing 23. Intercept method of a `HttpInterceptor` that adds the user's id token to all http requests if the requests URL matches an endpoint and a user is logged in.

## 4.4 Flask Back End

Flask can be installed inside an activated Python `venv` environment with the Python package installer with the CLI command:

```
pip install Flask
```

### 4.4.1 Blueprints and routes

Once flask has been installed, a simple flask can be set up with the code snippet seen in listing 1. Flask blueprints are used to create different application components. Blueprints are a set of operations a flask application will execute when started. Flask blueprints are used when developing flask applications that can grow large enough that the code base needs to be divided into multiple modules. A blueprint can be created with the following code:

```
bp = Blueprint('example_BP', __name__, url_prefix='/example')
```

This will create a blueprint named 'example\_BP', with all routes in the view having an URL prefix of /example. The blueprint can now be registered on the flask application with the `register_blueprint()` method. This will create the routes defined in the blueprint and register them on the application object.

As the translation application and management interface both are on different servers than the API, CORS needs to be enabled on the API. Otherwise, the user's browser would not send HTTP requests to the API. CORS can be enabled via the Flask-cors extension and with following code:

```
CORS(app, supports_credentials=True, resources={r"/*": {"origins": "*"}})
```

To create a route the `route()` decorator is used to bind a URL to a function. As HTTP requests use different HTTP methods when accessing URLs, the route can also define what HTTP methods have access to it. This is done with the `methods` argument of the `route()` decorator. If the `methods` argument is not defined, only `GET` methods will be allowed to call the function. The `make_response()` method is used to be able to send a response code, such as 400 and 401 if a request fails or if the user is not authenticated. To send JSON data as a response, the `jsonify()` method is used, which serializes a Python object into a JSON string using the applications JSON encoder and turns it into a response object. An example of a Flask route definition can be seen in Listing 24.

```
@bp.route('/test', methods=['GET'])
def test():
    return make_response(jsonify("message":"Here, have a cool message"),200)
```

Listing 24. Example of a simple route definition.

#### 4.4.2 Database and models

To query a MySQL database with Python, one can use the MySQL Connector for Python or use an ORM library. Using an ORM abstracts the writing of SQL queries into method calls on the data model. This way, the code base becomes more flexible as the same queries can be used on different types of databases. Excluding SQL from the application logic also makes it more readable. `SQLAlchemy` was chosen as the ORM library in the project of this thesis, as there is a flask extension available for it, which can be installed with Python package installer. An example of `SQLAlchemy` initialization can be seen in Listing 25.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
uri = 'mysql://root:password@localhost/test_db'
app.config['SQLALCHEMY_DATABASE_URI'] = uri
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
```

Listing 25. Example of a minimal Flask application with `SQLAlchemy` and a data model initialized.



Database models needed to be created for data that would be saved to the database. Two models were created, a translation model and a dictionary language model.

An entity of the translation model is an individual word in a particular language, such as in German or English. Each translation would have a field for the word, a usage note, an example sentence and the language of the translation. Translations that mean the same thing have the same word id. The translation entity also contains fields on its creation date and the information on the user who created it.

Dictionary language is a model for each language of the application. This model was added to allow the addition of languages without any the need to change application logic, as the languages a user can choose in the translation application is a query of all dictionary languages.

#### 4.4.3 Authentication

All API routes of the application had to be secured from unauthorized access. To validate that an HTTP request has access to a resource, it must include an id token in the authorization header of the request. The id tokens signature is validated using the Azure AD public key with Python's pyJWT cryptography module. As fetching a new private key on every request would add unnecessary delay to each request, the private key is saved to a JSON file. Due to security purposes, Azure AD's public key changes on a periodic basis and can be rolled over immediately, if needed. Due to this, the authentication method checks that the public key has not been updated every 24 hours and updates it, if necessary. The issuer and audience fields of the id token are also checked that they contain the domain and tenant ids. If an id token fails to validate, or a request does not contain an id token in its header, a 401 unauthorized status is returned to the user.

## 5 Summary

The purpose of this thesis was to develop a mobile-friendly dictionary application, which would allow users to translate railway-specific vocabulary from one language to another. The Ionic framework was used to develop the translation application, as this way both a web and a mobile version was able to be developed with HTML, CSS, and JavaScript.

A management interface was developed to manage the vocabulary and languages of the application. Both the translation application and the management interface use an API made with the Flask framework to view or make changes to the vocabulary. The thesis has covered the background, technology, design, and development of the application.

Developing the different application components was a gratifying experience, as I learned much about developing multi-platform web applications and hybrid applications. Learning about authentication was particularly gratifying, as that subject was entirely new for me when I started development on this project.

The Management interface of the application succeeded in fulfilling the requirements put upon it, as the management interface allowed the viewing, addition, editing, and deletion of translations in the application. The access to the management interface was restricted to authorized personnel, with the use of Azure Ad. The choice to use Angular as the development framework for the management interface proved itself to be the correct choice, as shifting from developing the translation application to developing the management interface happened without any problems.

Development of the translation application did not finish entirely during the creation of this thesis. Authorization was not successfully implemented on the translation interface in time to add its development process to this thesis. Development on the authorization continues after this thesis is completed. Excluding authorization, the application fulfills the requirements of being simple to use and available both in Finnish and English.

The API of the application fulfills the requirements of the application as it provides all the necessary API routes required to view and edit the data of the application. Development on the API also continues after the writing of this thesis, as some changes must be made to it to authenticate the HTTP request of the translation application. It was noticed during the development of this application that it would have been easier to develop the API of this application in a framework such as Node.js, instead of Python. This is because working in a single programming language (JavaScript) is easier than working with two different ones (JavaScript and Python).

## References

- 1 “Jargon – Definition and Examples of Jargon”. [Internet]. Literary Devices. 19 February 2014. [Cited 2019 April 3] Available from: <https://literarydevices.net/jargon/>
- 2 “Branding Guidelines for Angular and AngularJS”. [Internet]. AngularJS Blog. 27 January 2017 [Cited 2019 April 4] Available from: <http://blog.angularjs.org/2017/01/branding-guidelines-for-angular-and.html>
- 3 “Angular vs React Industry Trends: February 2019 Data Report” [Internet] Cloud Academy. [Cited 2019 April 5] Available from: <https://cloudacademy.com/research/angular-vs-react-industry-trends-february-2019-data-report/>
- 4 Google Trends. [Internet] React, Vue, Angular [Cited 2019 April 7] Available from: <https://trends.google.com/trends/explore?cat=733&date=today%205-y&q=React,Vue,Angular>
- 5 “Microsoft TypeScript: the JavaScript we need, or a solution looking for a problem?”. [Internet]. Ars Technica. 10 March 2012 [Cited 2019 April 4] Available from: <https://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/>
- 6 Architecture Overview. [Internet] Angular Docs. [Cited 2019 April 5] Available from: <https://angular.io/guide/architecture>
- 7 Introduction to components. [Internet] Angular Docs. [Cited 2019 April 5] Available from: <https://angular.io/guide/architecture-components>
- 8 Introduction to services and dependency injection. [Internet] Angular Docs. [Cited 2019 April 8] Available from: <https://angular.io/guide/architecture-services>
- 9 Angular. NgModules. [Internet] Angular Docs. [Cited 2019 April 8] Available from: <https://angular.io/guide/ngmodules>
- 10 Angular Tutorial. Services [Internet] Angular Docs. [Cited 2019 April 8] Available from: <https://angular.io/tutorial/toh-pt4>
- 11 Hazem Saleh. JavaScript Mobile Application Development. Packt Publishing. 2014. Chapter 1. An Introduction to Apache Cordova. Cordova architecture.
- 12 Cordova. Overview. [Internet] Angular Docs. [Cited 2019 April 9] Available from: <https://cordova.apache.org/docs/en/latest/guide/overview/index.html>

- 13 Ionic Introduction. [Internet] Ionic Docs. [Cited 2019 April 9] Available from: <https://ionicframework.com/docs/intro>
- 14 Flask Foreword. [Internet] Flask Docs. [Cited 2019 April 9] Available from: <http://flask.pocoo.org/docs/1.0/foreword/>
- 15 “2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use”. March 4, 2019. [Internet] ScaleGrid Blog. [Cited 2019 April 9] Available from: <https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/>
- 16 “Developer Survey Results 2018” [Internet] Stack Overflow. [Cited 2019 April 9] Available from: <https://insights.stackoverflow.com/survey/2018/#technology>
- 17 Introduction to SQL [Internet] W3Schools.com [Cited 2019 April 9] Available from: [https://www.w3schools.com/sql/sql\\_intro.asp](https://www.w3schools.com/sql/sql_intro.asp)
- 18 Holmes, Simon. Getting MEAN With Mongo, Express, Angular, and Node. Manning Publications. 2015. Chapter 3. Creating and setting up a MEAN project. Modifying Express for MVC
- 19 ASP.NET MVC Overview [Internet] Microsoft ASP.NET MVC Docs [Cited 2019 April 30] Available from: [https://docs.microsoft.com/en-us/previous-versions/aspnet/dd381412\(v=vs.108\)](https://docs.microsoft.com/en-us/previous-versions/aspnet/dd381412(v=vs.108))
- 20 Model-view-controller. [Internet] Wikipedia [Cited May 4] Available from: <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#/media/File:MVC-Process.svg>
- 21 “ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET” [Internet] MSDN Magazine Blog November 2013. [Cited 2019 April 30] Available from: <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>
- 22 Single page application advantages and disadvantages. [Internet] Zymphonies blog [Cited 2019 April 30] Available from: <https://www.zymphonies.com/blog/single-page-application-advantages-and-disadvantages>
- 23 Panhale, Mahesh. Beginning Hybrid Mobile Application Development. Apress 2016. Chapter 1. Introduction to Mobile Application Development Ecosystems.
- 24 “Why did we build Visual Studio Code?” [Internet] Visual Studio Code docs. [Cited 2019 April 30] Available from: <https://code.visualstudio.com/docs/editor/whyvscode>

- 25 “Requirements for Visual Studio Code” [Internet] Visual Studio Code docs. [Cited 2019 April 24] Available from: <https://code.visualstudio.com/docs/supporting/requirements>
- 26 “Developer Survey Results 2019” [Internet] Stack Overflow insights. [Cited 2019 April 24] Available from: <https://insights.stackoverflow.com/survey/2019#overview>
- 27 “Theming your Ionic App” [Internet] Ionic Docs [Cited 2019 April 24] Available from: <https://ionicframework.com/docs/v3/theming/theming-your-app/>
- 28 Angular Tutorial. Routing. [Internet] Angular Docs [Cited 2019 May 2] Available from: <https://angular.io/tutorial/toh-pt5>
- 29 Routing & Navigation [Internet] Angular Docs [Cited 2019 May 2] Available from: <https://angular.io/guide/router#routing--navigation>
- 30 Router Links [Internet] Angular Docs [Cited 2019 May 2] Available from: <https://angular.io/guide/router#router-links>
- 31 Getting Started [Internet] Angular Material Docs [Cited 2019 May 2] Available from: <https://material.angular.io/components/table/overview>
- 32 “What is authentication?” [Internet] ADD docs [Cited 2019 May 5] Available from: <https://docs.microsoft.com/en-us/azure/active-directory/develop/authentication-scenarios>
- 33 “Understanding the OAuth2 implicit grant flow in Azure Active Directory (AD)” [Internet] ADD docs [Cited 2019 May 05] Available from: <https://docs.microsoft.com/en-us/azure/active-directory/develop/v1-oauth2-implicit-grant-flow>