Technische Universität Braunschweig

Institut für Nachrichtentechnik

Bachelor Thesis

Development of an Administrative Web Frontend for Deep Learning Research

Lukas Güldenhaupt

Matrikelnummer: 4571429

01.03.2017

Technische Universität Braunschweig
Institute for Communications Technology
Schleinitzstraße 22 – 38106 Braunschweig

Prüfer: Prof. Dr.-Ing. Tim Fingscheidt
Betreuer: Samy Elshamy, M.Sc.

Erklärung

Hiermit versichere ich die vorgelegte Bachelor Thesis zum Thema

„Development of an Administrative Web Frontend for Deep Learning Research“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorarbeit zum Thema „Development of an Administrative Web Frontend for Deep Learning Research“ selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln erstellt habe.

Braunschweig, den 01.03.2017

_____

Lukas Güldenhaupt

Abstract

Keywords:

Contents

Content **Page**

1 Architecture

Since the basic idea of this tool is to give a lasting web frontend for the institute a good choice of what software to use is essential. Therefore, for client and server-side code we chose well maintained and well-known frameworks as there are Meteor as a mostly server-side JavaScript framework and Angular as a frontend JavaScript framework. With this an all in all forward-looking webpage is ensured. In this chapter we evaluate why the chosen software fits our purpose and how they work together. We could build the web server and client completely from scratch but Meteor and Angular provide an overall good structure and a solid base for further development. Furthermore, we chose MongoDB as a database.

1.1 Server-side

As mentioned before Meteor is our chosen framework for the server side. It is an opensource full-stack JavaScript platform for web, mobile and desktop development. The power of this platform is its fast learning curve, its usability for any device and its technology integration. What that means is that without knowing much about webservers you can easily create your own application. Meteor also is known for its compatibility since you can use it independently from the platform, no matter if its web, iOS, Android or desktop. In our case we use it as a webserver but with further development of the website it could be optimized for mobile devices or become an app itself if desired.

A big point for Meteor is that you can share code between server, client and the database, what accelerates the development process enormously. This is what makes our application very reactive. Meteor uses data on the wire, sending not HTML to the client but data which gets rendered directly on client-side. With that provided reactivity the client reflects the true state of the data in real time. In combination with our frontend framework Angular no page reloading is necessary to have the latest data.

Behind the easy-to-use platform lies a NodeJS-server. When deploying Meteor-code it generates a stand-alone Node application. And this is the only dependency it has which means everywhere you have NodeJS installed you can run your meteor application.

1.2 Client-side

On client-side we chose the JavaScript framework Angular in version 4, developed and maintained by Google. Angular makes client development across all platforms possible. It grants fast speed and good performance and allows us to extend the template language HTML with

our own written components. Nearly every web IDE supports Angular to give the user syntax-highlighting, code-completion and Angular-specific help. In our case it replaces the Meteor-standard blaze-templates. Meteor and Angular work perfectly together on various platforms, when displaying real time data and keeping the reactivity of our application on a very high level.

1.3 Database

We chose MongoDB as our database. It is a strong and popular no-SQL, document driven database. Even with large data sets it scales very well and provides a high performance. Unlike in SQL an entity is represented by a collection which contains documents as its entries. A document is very similar to a JSON-Object and can be easily read and modified. Thanks to the flexibility of MongoDB we can design our collections freely and edit them with small effort, without losing our existing data. We can define the basic structure of a document and shape the rest of it as we want to.

This feature comes in handy when we have very variable data entries. In our case it does not matter how a given configuration file generated by one of many neural network programs looks like. With MongoDB we could insert the data without adjusting it to match a pattern.

Another big point of using MongoDB is that you have a decreased learning curve. Making queries is easy to understand and use. The necessary concept of using foreign keys to connect documents is also featured as every entry has its own unique id.

2 Software Design

In this chapter we explain how our web-application is generally structured, how and where the different tasks are handled. Certain constructs are set by the technology we are using however there are a lot of conceptual thoughts to be made. For example, how the code should look like to be intuitive on the one hand and compact at the other. Very important is the fact, that the progression does not have to be finished with the work of this bachelor thesis. The application is build and meant for further development. Therefore, a good documentation and clear project structure is helpful.

At first, we declare the design choices made on the client-side of our application. Later, we get to the data-handling and the server-side structures.

2.1 Typescript

Both our client and our server almost fully consist of JavaScript code. In fact, we use ECMAScript 5, which is a standardized version of it. The syntax of JS is similar to C or Java, which makes it easy to understand for everyone who has coded a bit. There is one big downside to it, being type insecure. Pure JS has no variable types. In ES5 to declare a variable, you can choose between *var*, *let* and *const* as a keyword. Each keyword has a different function or scope. To declare a variable globally you chose var, to declare a variable scoped between to curly brackets let should be used and to declare constant variables, that will or must not change const is the keyword to go. These keywords are helpful in some way but when it comes to huge applications with data-handling and complex functions a better approach is needed. Fortunately, Angular uses an extension called Typescript to ensure better coding qualities. Typescript compiles to JS what then can be seen in the browser. It supports definitions of classes, interfaces, generics, enumerations, inheritance, types of course and more useful features. With a package for Meteor we can write even our server code in typescript bringing this java like structure to the whole project. With Typescript the code is much more readable, clearer and java like.

2.2 Components

Angular offers a system to encapsulate logically independent code. These blocks of functionality are called “NgModules”. The root of our web application is the AppModule which contains all of our classes, services and helpers.

With Angular we can create UI segments called components. A component has a visual part and a logical part. You could say, that an Angular application is a tree of components. This could be a whole page, a table, even a text label or anything you want. Thanks to the independence of a component you can as many instances as you want anywhere in your application.

We built our web page as a single page with one module which contains one master component, the *AppComponent*. All other components are children of this. This has the strategic benefit of every page looking similar. With that we can have a head navigation and basic menu features, no matter what other component is loaded currently. In extension to that, there are no big interruptions when switching the view, because it all relies on the same base module and component.

2.2.1 Basic Component Structure

We decided to store all component parts in a single folder to keep the overview. Components consists mostly of three parts: template, style and the component itself written in TypeScript. The template written in HTML defines the basic structure of the view. Together with the style written in SCSS it unites to a designed website and the code gives its functionality. Thanks to Angulars component construct the application can be easily extended by further features.

2.2.2 Routing

The *AppComponent* contains the headline navigation and a router outlet. This outlet is a place holder for any component, we want to navigate to in our application. A good way to deal with routing is the Angular basic package angular-router. We can easily define routes and their corresponding component and even add so called guards to grant access control over the routes. A route path is the name of the route you need when you want to navigate to the view as well as the additional part of the URL in the browser. For example, if the view shown is the dashboard, where you can manage and navigate to your projects, the route defines the path as "/dashboard" and the component to the *DashboardComponent.* To show the view simply navigate to it with code or add the path to the URL. This way we can have the benefits of a single page website without losing control over the navigation.

A great feature is to add dynamic parameters for each path in a route. This is useful when showing the page of a specific project or configuration. By adding an ID to the path for example, we can use the same component for every entry but having distinct content on the

view. When sharing links to your project, a configuration or a mapping the identity of it is stored in the URL.

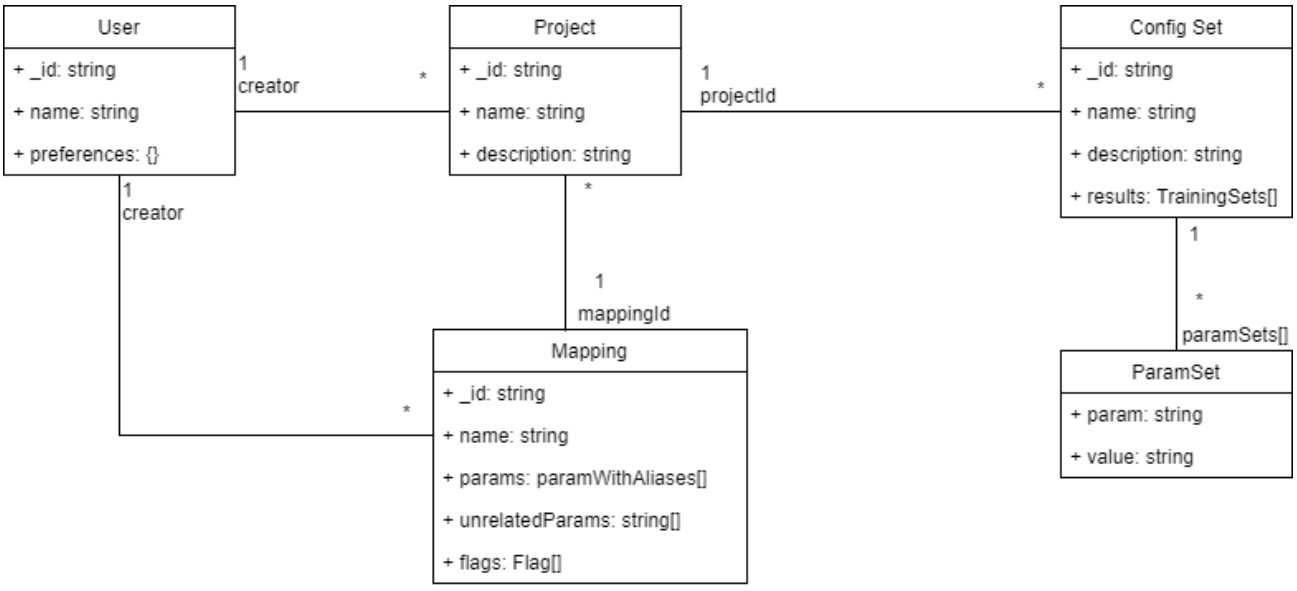
As mentioned before we can secure our website or single components with guards. Guards get called whenever someone tries to go to a different page or view. In our application all Components besides the *LoginComponent* are not accessible when the user is not logged in.

2.3 Data-handling

There are two important parts to differentiate when talking about the data-handling in the project. The first part is the data stored in the database, which we extract using services. The second part is the design of the individual types and classes we created. When getting the data from the database we cast them to match our types, classes or interfaces. In this chapter we take a look at the data model lying behind our database, how we distribute the data between our components and at Observables, which are a powerful extension from another package called RXJS.

2.3.1 Data model

Our chosen database, MongoDB, is a NoSQL solution as earlier mentioned. Therefore, the data model is a bit more abstract and does not represent the stored data equivalently. The support of query based collection joins is limited, so that these tasks are handled by the application itself. This way we also guarantee more straight forward code which is better to understand. In the following section we will explain how the data we need to persist in our application is stored.



There are a few entities that relate to collections in our database. First of the user collection. Meteor offers this collection with some basic features like usernames, mail address and so on. For our cases a username and an id are all we need from that. Additionally, we provide preferences to every user, so we can store things like last chosen options for filters, dynamic tables or even design preferences. The user's preferences are an object completely designed by our client-side application, easy to modify and adjust.

The one big entity we are building around is a ConfigSet. ConfigSets will be created when uploading a configuration file, containing parameters and results. The parameters are an array of a parameter name and a value for that name. If the application finds any results they will be split into training sets and stored as an array of numbers.

Together with the extracted information an id, the creator's id, a name and a description are saved in the ConfigSet collection as a document.

For better management of the configurations, every ConfigSet is related to a project, which can be created by any user. A project has a name, an id, an optional description, the creator's id and a mapping id. With the creator's id we can ensure that every user has his own projects and configuration files, where no one else can manipulate or delete his work.

As mentioned a project can relate to a mapping, which essentially maps parameters to other parameters, by defining aliases. So, a parameter can have multiple names. This is needed to filter or compare between two configurations of different sources or programs. A mapping stores the creator's id for later access control. Furthermore, the related and unrelated parameters are stored.

Besides the functionality to declare aliases a mapping can contain flags, to translate the values of parameters. A user can define his own flags in his mapping for any value.

2.3.2 Data services

For every collection in our database we have a data service in our application to control the data flow. A data service handles the queries and distributes the documents to the components. The most common queries are those to create a new document and update or delete an existing one. Every data service has a reference to the collection, for example the ConfigSet

data service has a reference to the config-set-collection. The client as well as the server are aware of all collections. However, the queries are only made on client side.

When the user creates a new ConfigSet by adding a configuration file to a project, the ConfigSet data service will call the query to create a new document. When the document was successfully created the MongoDB will return the id of the new ConfigSet, which then will be returned to the application to inform the user about the success or failure.

Because most of the data base actions are asynchronous the data services will often return observables to keep track of the progress.

2.3.3 Observables

Observables are powerful constructs to provide asynchronous information. As previously mentioned data services make use of those when fetching data or performing other queries. We use RXJS as a package to have access to observables, subjects, iterators and many other useful tools. An observable will call functions like getting every document of a collection. This is an asynchronous job, because that can take time if the data base is very busy. When subscribing to that observable every time a new document was found, the application can react to those.

2.4 Authentication

The whole application will be exclusively available for employees at the institute for Communications Technology or those who have an account at their LDAP system. To acquire this, every user has to login with their institute credentials first. For this feature another Meteor package called *accounts-ldap* is needed. With this every time a user performs his first login and the LDAP system confirms the successful authentication meteor creates a new user at the users-collection. On furthers logins this user document will be used again. There is now other way for creating a new user. This way we can ensure that whenever the client knows the user's identity, this user is authorized to work with the application. As mentioned before we can lock the routes to every component with guards. The main guard in our website checks whether a user is logged in or not and restricts or grants access to the pages.

3 Users perspective

3.1 Profile/Login

3.2 Projects

3.3 Configurations

3.3.1 Mappings

3.3.2 Filtering

4 Developers Perspective

4.1 Developer Tools

4.2 Setting up a development environment

4.3 Coding

4.3.1 Adding Components

4.3.2 Adding Collections

4.3.3 Working with Observables

4.3.4 Extending Functionality

-Npm packages

4.4 Documentation

4.5 Deployment

A Appendix

B Bibliography

Im aktuellen Dokument sind keine Quellen vorhanden.

C List of Figures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

D List of Tables

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

E Abbreviations

E

ES5 · *ECMAScript 5*
ES6 · *ECMAScript 6*

J

JS · *JavaScript*