Bachelor Thesis

Development of an Administrative Web Frontend for Deep Learning Research

Lukas Güldenhaupt

Matrikelnummer: 4571429

16.02.2017

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 16.02.2017

____

Lukas Güldenhaupt

Abstract

The application presented in this bachelor thesis is named ANNA, which stands for ‘Administrative Neural Network Application’. It is a web frontend for deep learning research and provides the functionality to store and manage configurations and results of deep learning programs. Filtering the entries is a main feature and allows the user to keep overview of his work. The application was made using several web frameworks, as there are MeteorJS [**Met17**], NodeJS [**Nod17**], and Angular [**Ang17**]. It uses a MongoDB [**Mon17**] as database. Presented are the architecture, the chosen software design, the user’s and the developer’s perspective of the web frontend.

Keywords: deep learning research, administrative web frontend, web application, Meteor, NodeJS, Angular, MongoDB

Zusammenfassung

Das in dieser Bachelorarbeit vorgestellte Programm heißt ANNA, was für „Administrative Neural Network Application“ steht. Es ist eine Webseite für Deep Learning Forschung, die die Funktionalitäten bietet, Konfigurationsdateien und Ergebnisse von verschiedensten Programmen zu speichern und zu verwalten. Das Filtern der Einträge ist dabei eine Hauptfunktion und erlaubt es dem Benutzer, den Überblick über seine Arbeit zu behalten. Die Anwendung wurde mithilfe der Web-Frameworks MeteorJS [**Met17**], NodeJS [**Nod17**], und Angular [**Ang17**] erstellt. Zur Speicherung der Daten wurde eine MongoDB [**Mon17**] gewählt. Im Zuge dieser Arbeit werden die Architektur, das Software-Design, die Nutzer-, sowie die Entwicklerprespektive der Webseite beleuchtet.

Keywords: Deep Learning Forschung, Administrative Webseite, Webanwendung, Meteor, NodeJS, Angular, MongoDB

Contents

1 Introduction

When solving problems using deep learning, there is a variety of tools available to work with. The parameter set differs from program to program and makes it difficult to keep track of the output and input data. Moreover, the ability to search for the best results or for a certain set of parameters can be uncomfortable if more than one deep learning research tool was used. Oftentimes, there are hundreds of trained networks with different parameter sets and with that, different results.

With the work of this thesis, an administrative web frontend was developed, to manage the projects and configuration files of any kind. It allows a permitted user to organize his configuration files and results in projects. Furthermore, it provides the functionality to equalize the parameter sets of different deep learning tools, so that it is easy to filter and search through the produced data. The application will help to improve the tracking of project related files and takes care of inequivalent data input.

Created was an application that should be used as a basic. With the chosen architecture and tool set that the web frontend was developed with, it is easy to extend its functionality.

This thesis is separated into six chapters. After this introduction, the used architecture and tool set is explained, talking about the server and the client side, the database and the composition of all the components together. In the third chapter we focus on how we handled certain parts of the software design. This includes the programming language we chose, the basic structure of the used frameworks, data handling and the authentication.

After talking about the structural part of the application, the user's perspective is presented. This chapter works as a guide on how to use the web application and explains all the views and features.

The fifth chapter is about the perspective of a developer and should help to enhance the tool with further development. It takes care of all technical things a developer needs to know when extending the application, concerning the setup of the developer's environment, the basics of the used frameworks, the documentation and the deployment. At last, the conclusion is presented.

2 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 [**Met17**] as a mostly server side JavaScript (JS) framework and Angular [**Ang17**] as a frontend JS 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 [**Mon17**] as a database, which is explained later (see 2.3).

2.1 Server Side

As mentioned before, Meteor is our chosen framework for the server side. It is an open-source 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 it’s a web, iOS, Android or desktop application. 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 advantage of Meteor is that you can share code between server, client, and the database, which accelerates the development process enormously. This is what makes our application very reactive. Meteor uses data on the wire, sending not Hypertext Markup Language (HTML) to the client, but data which is rendered directly on the client side. With the provided reactivity the client displays the true state of the data without any delay. In combination with our frontend framework Angular, no page reloading is necessary to obtain the latest data, as it gets refreshed on every data change.

Behind this easy-to-use platform lies a NodeJS [**Nod17**] server with its package system, the NodeJS-Package-Manager (NPM) [**NPM17**]. When deploying Meteor code, it generates a standalone NodeJS application. This is the only dependency it has, which means everywhere where NodeJS is installed, a meteor application can be executed.

2.2 Client Side

On client side we chose the JS framework Angular in version 4, developed and maintained by Google Inc. [**Goo17**]. 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 integrated development environment (IDE) for web development 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, while displaying data, without delay or loading and keeping the reactivity of our application on a very high level. With the complete tool chain, the application can be seen and used on every up to date browser.

2.3 Database

We chose MongoDB as database. It is a strong and popular no-SQL, document driven database. Even with large data sets, it scales very well and provides 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 JavaScript Object Notation object (JSON-Object) [**ECJ17**] 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 adjust 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 can insert the data without adjusting it to match a predefined pattern.

Another advantage of using MongoDB is that it is easy to learn. 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.

2.4 Architecture Composition

To better understand the composition of the different parts and frameworks playing together, a visual illustration can help. Figure 1 shows a simplified version of the client and server side parts as well as the connections they use.

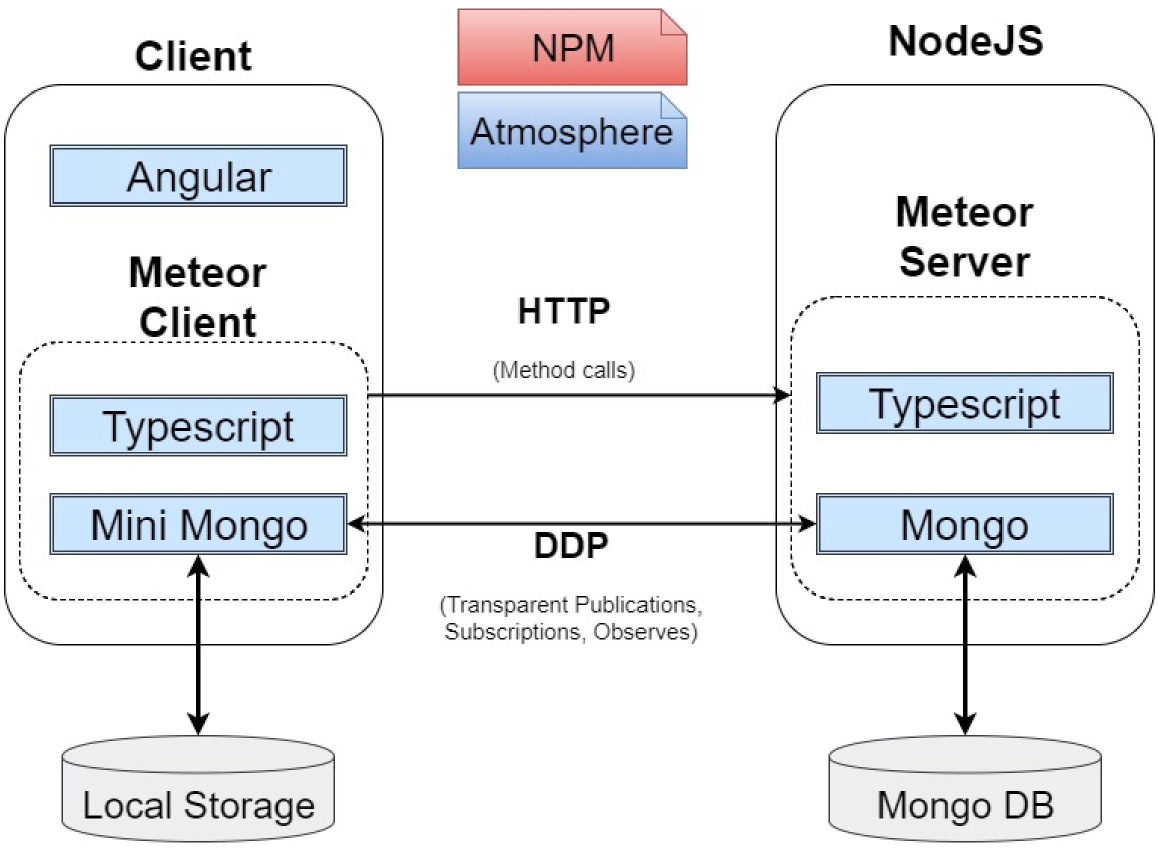

Figure 1 – Simplified Architecture composition. Client side with Angular and Meteor part on the right side. The Meteor client side stores the data from MiniMongo on the local storage. The server side with Meteor using Typescript and Mongo to connect to the Mongo database.

In this figure the server side on the right is titled with NodeJS, because it is the main framework our application is built on. Meteor abstracts the low-level framework by adding a more comfortable handling. It also makes the server more reactive, as every client is in sync with the server and refreshes automatically when new data is available. In our application we can use Typescript, which is explained in section 3.1 Typescript, on the server side as well as on the client side. Because we use MongoDB, a Mongo package comes with Meteor to handle the connection to the database.

The client side also has a Meteor component to keep the reactivity. In order to show the same data pool the server can see, a MiniMongo is part of the Meteor client. It performs the same actions the server does on the real database, but stores them at the local storage. The MiniMongo is a copy of the real database and always in sync with the real database. As shown in the figure, the client's MiniMongo and the server's Mongo communicate via Meteor's Distributed Data Protocol (DDP) to synchronize each other. This DDP is a very valuable feature and another reason why we chose Meteor. Meteor's client side uses the standard HTTP posts and gets to make explicit method calls.

On top of the client, we use Angular, which perfectly makes use of the reactivity Meteor provides. Angular also works with Typescript.

On both sides we have the NPM to extend functionality with existing packages. In addition to that, Meteor has a website for Meteor specific packages, called AtmosphereJS [**AJS17**]. Using existing packages will be explained in the section 5.7 Extending Functionality with NodeJS.

3 Software Design

In this chapter we explain how our web application is structured in general and how and where the different tasks are handled. Certain constructs are set by the technology we are using, like components or modules, which are explained further in this section. 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 on the other. Very important is the fact, that the development 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 and required, to allow future developers to easily enhance the framework.

At first, we introduce the design choices made on the client side of our application. Later, we continue with the data-handling and the server side structures.

3.1 Typescript

Both, our client and our server almost fully consist of JS code. In fact, we use ECMAScript 6 (ES6) [**ECM11**], 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 some experience in coding. There is one big downside to it, being type insecure. Pure JS has no variable types. In ES6 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 [**Typ17**] to provide an improved programming environment. Typescript compiles to JS, which then can be interpreted by 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 closer to Java.

3.2 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. Every module has a configuration file, where the routes, the declarations for components, providers like services and other imported modules are configured.

With Angular we can create user interface (UI) segments called `Components`. A `Component` has a visual part and a logical part. Regarding an Angular application, it 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 create as many instances as you want anywhere in your application.

We built our web application as a single page with one module, which contains one master `Component`, the `AppComponent`. All other `Components` are children of the `AppComponent`. This has the strategic benefit that styles get inherited and pages obtain a unified appearance. 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`.

3.2.1 Basic Component Structure

We decided to store all `Component` parts in a single folder to keep the overview. `Components` consist 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 Syntactically Awesome Stylesheets (SASS) [**CNE17**], an improved version of Cascading Style Sheets (CSS), it unites to a designed website and the code gives the functionality. Thanks to Angular's component construct the application can be easily extended by further features.

3.2.2 Routing

The `AppComponent` contains the headline navigation and a router outlet. This outlet is a placeholder 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 Uniform Resource Locator (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 `Dashboard-`

`Component`. To show the view, simply navigate to it with code or add the path to the basic URL of the server. 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 identifier (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`, their identity 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.

3.3 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 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 Reactive Extensions (*RXJS)* [**RJS17**].

3.3.1 Data Model

Our chosen database, MongoDB, is a NoSQL. NoSQL is an alternative for the known Structured Query Language (SQL) databank system, which has less restrictions and is more adaptable then the conventional SQL. 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 can guarantee more straightforward 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. Therefore, shows the data model diagram.

There are a few entities that relate to collections in our database. First of all the user collection. Meteor offers this collection with some basic features like usernames, mail address and more. For our cases a username and an id are all we need from that. Additionally, we provide preferences to every user, so that 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.

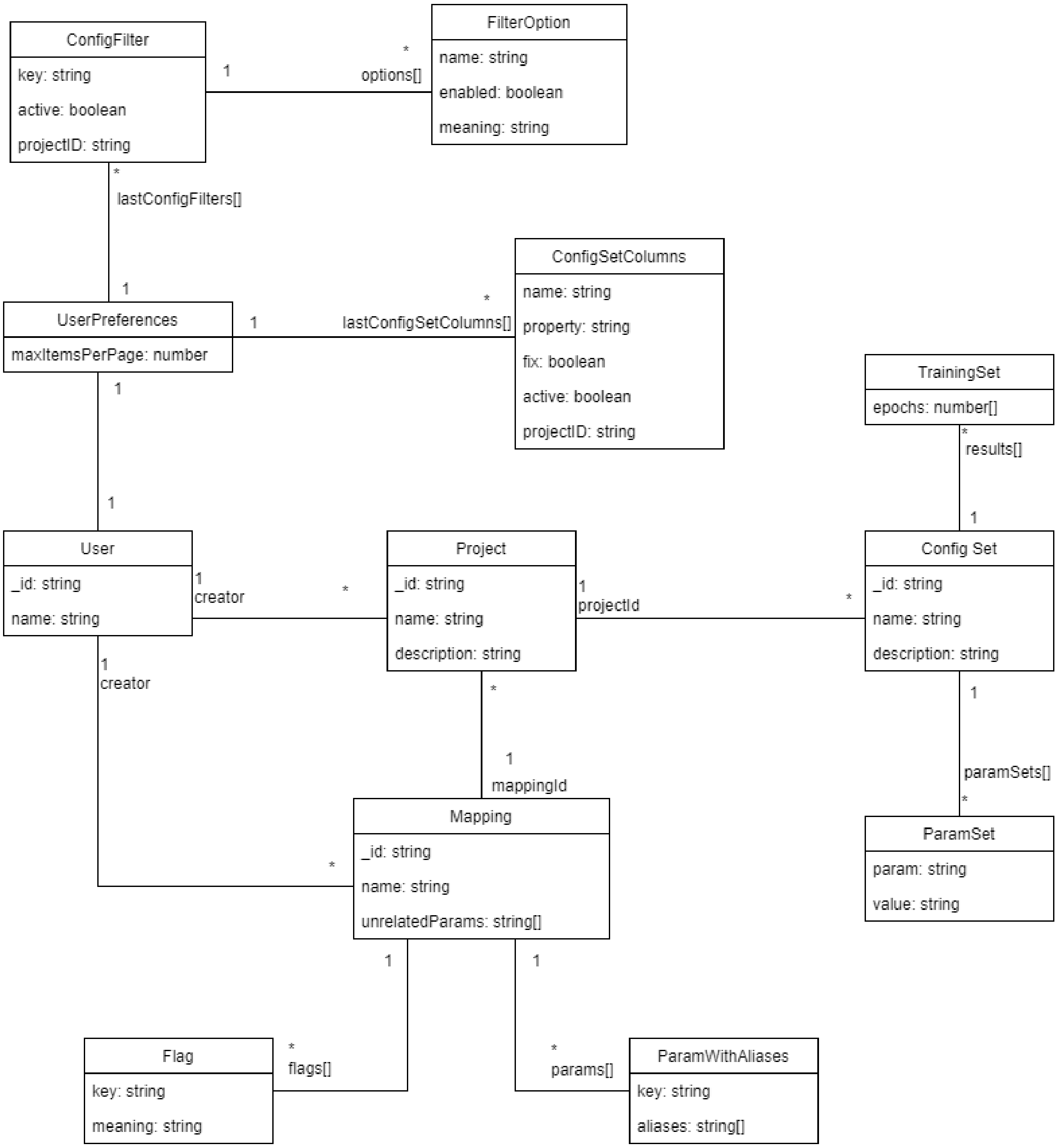

Figure 2 - The complete abstact data model. Showing only attributes, no methods. These entities are stored in the database. During application runtime other temporary entities might exist, which are not persisted.

The most important entity we are using is a `ConfigSet`. `ConfigSets` will be created when uploading a configuration file, containing parameters and results. The parameters are an associative array storing each parameter name and the corresponding value for that key. If the application finds any results of the training process they will be split into spearate 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 already mentioned, a project can relate to a `Mapping`, which essentially maps parameters to other parameters, by defining aliases. Thus, a parameter can have multiple names. This is required 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.

3.3.2 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 `ConfigSet` collection. The client as well as the server are aware of all collections. However, the filtering of the data is mostly made on client side. There are some things like only returning user specific data, like private projects, which are handled directly on the server, side query to ensure security.

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.

3.3.3 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 is found, the application can respond to it.

3.4 Authentication

The whole application will be exclusively available for employees at the institute for Communications Technology or those who have an account at their Lightweight Directory Access Protocol (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 no other way of 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.

4 Users Perspective

In this chapter the application will be explained from a user's perspective. It should be a guide on how to use and where to find the functions. We start by introducing the general functionality, like creating a project or uploading configuration files and results, and will continue by getting more into detail. Before describing the application, there are a few terms to explain, which are common in modern web language. Also, the main structure of the application is presented.

4.1 Technical Terms

4.1.1 Toast

To inform the user about the success of actions or to give a short notification, a toast is displayed. Toasts are small cards often appearing at an edge of the browsers window, containing a short piece of information, like 'Empty Password' (see Figure 3) or 'Successfully created'. These toasts last for a few seconds and can be dismissed by dragging them to the side.

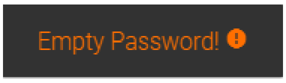

Figure 3 – Example toast with error message. Toasts are often shown in the upper right corner as notifications.

4.1.2 Modal

Another construct is a modal, which is mostly used as a dialog box, for confirmation messages or small content to show, without leaving a page. When a modal opens, the background gets dimmed and a card will pop up, containing some information. Some modals can be dismissed by clicking on the dimmed background, causing nothing to happen. Others containing more important information, like confirming a delete action (see Figure 4), can't be dismissed.

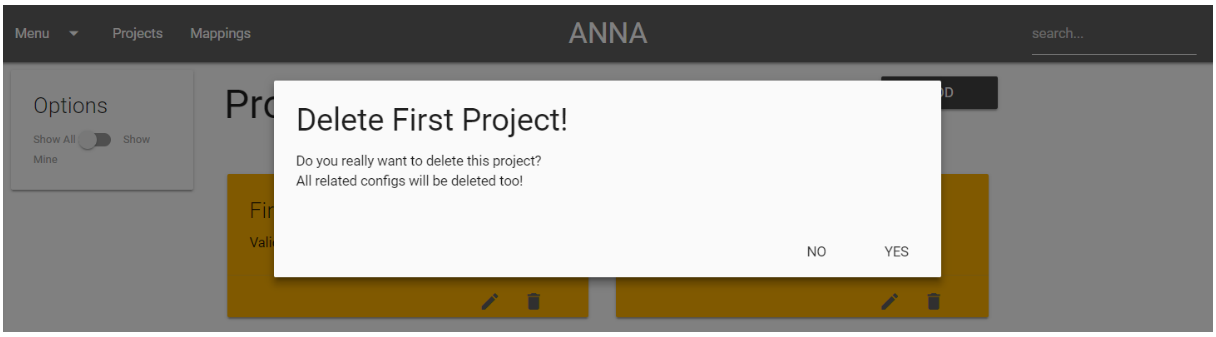

Figure 4 – Confirmation modal for deleting a project. Dimmed background. Two buttons for closing the modal, one for accepting the deletion, one for aborting the action.

Those modals often contain two or more buttons, giving the user options to close the modal. There is one modal opening, when the user uploads files. This one also cannot be dismissed and automatically closes once the upload is finished.

4.2 Basic Page Structure

Every page of the application shares some elements but also has its unique content. To navigate through the pages, the menu bar at the top of every page is a useful tool (see Figure 5).

Figure 5 – The menu bar on a desktop browser. Contains links to the dashboard and mappings, a search bar and a drop down menu.

It is a fixed bar containing links to the project dashboard and the `Mappings` page as well as a dropdown menu for actions like log out. On the right of the menu bar there is a search form, which can be used for filtering projects, configuration files or parameters on their specific pages. When the application is shown at a mobile device, the menu bar transforms, so that only the title and one button on the left remains. The button on the left opens a side menu known from mobile applications. This side menu also contains the links and the search form.

Below the menu bar, the specific page content is displayed. Every page has three columns, containing the main information in the middle and actions to perform or additional information on the sides.

4.3 Profile/Login

As explained earlier, the application is only accessible for those, who have an account at the institutes LDAP system. In order to use any function or see the work of others, a user has to log in first. When visiting the website, the user gets automatically redirected to the log in page as shown in Figure 6.

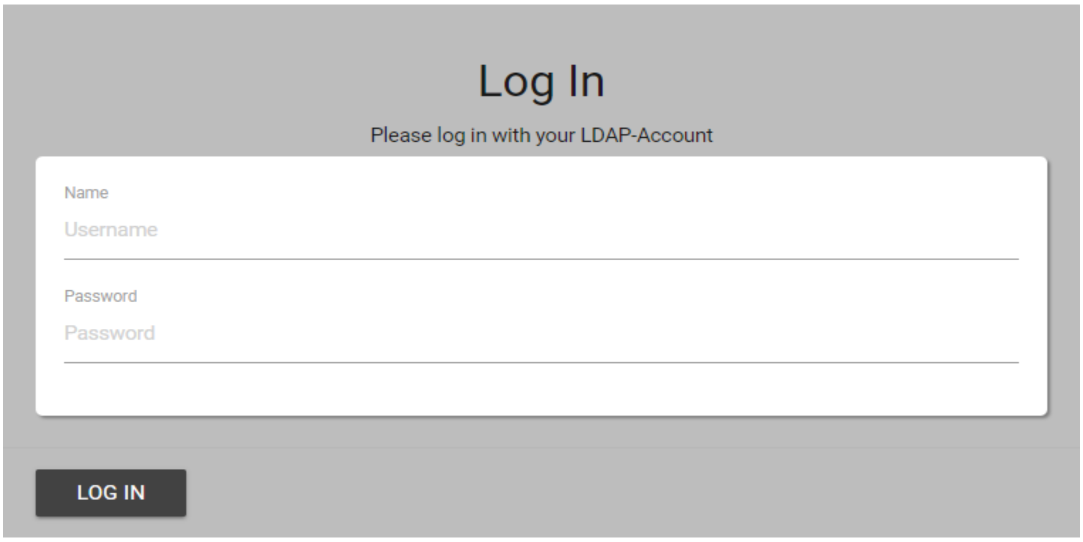

Figure 6 – The log in page with username and password form and log in button. This is the main view for the LoginComponent, which the user will get redirected to, if he is logged out.

Here, the user can enter his username and password. After clicking on the log in button, the user gets notified about the result of his operation in form of a toast. If the log in was successful, he gets redirected to the project dashboard. If not, the error message is displayed.

The account bound to this user is very important. Only the creator of a project, `Mapping` or configuration file can edit or delete it. Any other users may see the public projects but cannot manipulate the data of others. In addition to those rights, any stored preferences are also bound to the user, which we will learn more about later.

To log out, the user can click on the menu button on the top left corner and press the `LogOut` button. If the log out was successful, he again gets redirected to the log in page. Because this function is bound to the menu bar, it will always be accessible, no matter on which page the user currently is.

In the following sections we assume, that the user is logged in.

4.4 Projects

The main page and the first page a user needs is the project dashboard (see Figure 7). The dashboard gives an overview of all the projects. New projects can be created, and existing projects can be edited or deleted on this page.

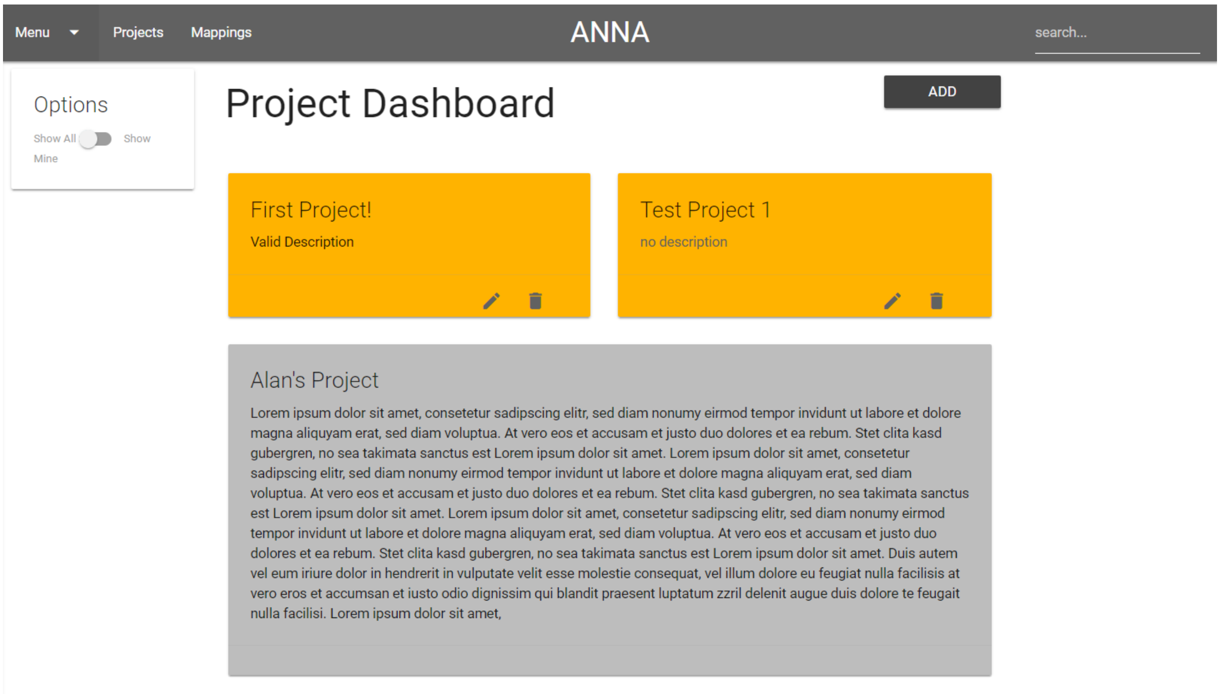

Figure 7 – The project dashboard with three project cards, two own and one public project. This is the DashboardComponent. One button for adding projects. Clicking on a project will redirect to the project's page. Projects can be filtered with the 'show all' toggle or the search bar.

A project is like a folder for configuration files. It has a name and an optional description and contains all the configuration files related to that project. Because every configuration file needs a project, it is essential to create one project first before uploading files. Creating a project is a short procedure. On the top right corner of the dashboard page is the add project button. When pushing the button, a modal opens (see Figure 8).

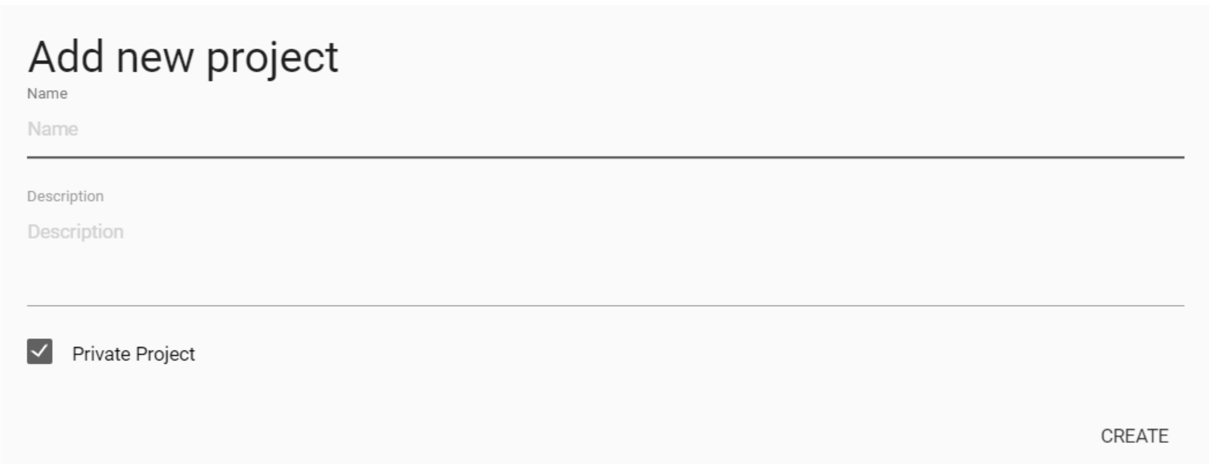

Figure 8 – The modal to add a new project with name and description field, the visibility checkbox and the create button. Similar to the edit project modal.

The user can now enter the name of the project as well as an optional description. The description is a place to share the intention of the project or leave information about important things everyone can see on first sight of the project. The name is required and without it, the project cannot be created. The modal holds the option to either create a public or a private project. Public projects can be seen by everyone, private ones are only accessible by the owner. This means, that the project and every project's configuration files is protected. The creation is completed once the user presses the create button. If the project was created in the database, a success message displays and the modal closes. Otherwise the modal stays and an error message is shown. This can happen, if for the example the database connection is lost, or no name was entered.

If the project was created, it immediately appears at the project dashboard page as a card as shown in Figure 9.

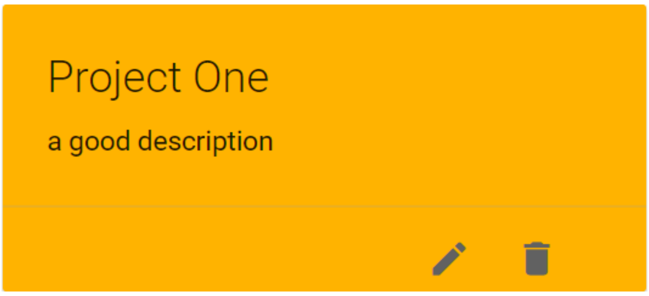

Figure 9 – A project card at the dashboard page, containing the name, the description and buttons for editing or deleting the project (only visible for the owner of the project).

There are two buttons on this card, the first one to edit the name and the description, and the second one for deleting the project and all of the configuration files it potentially contains. In order to change the project's name, description, or visiblity, the user has to click on the edit symbol. After checking if the user is the owner of the project, and thus allowed to edit it, another modal will open, which looks similar to the creation modal, where the attributes can be modified and saved. If the user is not allowed to edit the project, an information toast will be displayed saying, that he is not permitted.

When deleting a project, the user needs to click on the delete symbol and confirm his action on the confirmation modal (see Figure 4). When he confirms the deletion, the project and all of the related configuration files will be deleted from the database. There is no way to undo this operation.

To filter the projects on the dashboard page, the user can use the search form at the menu bar. While typing, those projects where neither the name nor the description matches the search text, will disappear from the dashboard. Another way of filtering is to either show only the owned projects, or every accessible project. This can be done by toggling the switch on the top left corner of the dashboard page.

The next step is to upload configuration files, which can be done in any owned project. To go to a projects page (see Figure 10), the user can click on the referring card at the dashboard. On a project's page the configuration files can be uploaded, seen and filtered, also a `Mapping` can be created, assigned, updated or removed from the specific project.

Figure 10 – The project page, showing the stored ConfigSets at the centered table. Also showing the card to add filters, the card to upload configurations and the card for the related Mapping.

4.5 Configurations

The configuration files can be uploaded by dragging them onto the *Drop Config Card* or clicking the upload button at a projects page (see Figure 10, upper right corner). The upload of multiple files at once is allowed. When uploading files, a preview of the results will be presented in a modal. Once all files are parsed and uploaded to the database, the user can close the modal.

If the application is able to find parameters with their values and results, the configuration file will be added as a `ConfigSet` to the database. By default, the uploaded file name will be chosen as the `ConfigSet` name and the file creation date is used as a description. The name and description can be edited later at any time, when visiting the `ConfigSet` page. Every `ConfigSet` can be deleted by clicking on the respective delete button (trash can symbol).

4.5.1 Valid Configuration Files

Valid configuration files must only contain plain text. Their parameters with referring values should be the first line. The application will accept only parameter-value pairs of the form `-parameter value` or `--parameter value`. Each pair must be separated by a whitespace.

All following lines represent the results. There can be as many data sets as needed with limitless epochs. One line of the configuration file represents one epoch. The training sets are separated with commas.

A valid configuration file could look as shown in Figure 11.

```
-max_epochs 50 -batch_size 100 -seedShuffle 101 -shuffleBetweenEpochs 9
0.55480844104033,0.60141967420318,0.51258117675424,
0.54117906510319,0.5950896572492,0.51669055504956,
0.53417116743207,0.5938269419786,0.51746665658443,
0.52928854141669,0.59668701752116,0.52390539328913,
0.52541625978031,0.59623656420042,0.52046824144995,
0.52282316837224,0.59889026328686,0.52766177420137,
0.51987304333549,0.60338957944663,0.54502123089387,
0.5177415907018,0.60728226026097,0.54852213012404,
0.5143780298571,0.60679450689978,0.54913152055047,
0.51283156338292,0.61077427045316,0.5536620043207,
0.51123206029802,0.61371967231147,0.55627606824778,
0.50956557843006,0.61716544343869,0.5612538174532,
0.50731329156632,0.61969332392649,0.56401290475339,
0.50680860050208,0.62255160168668,0.57893489267694,
0.50529190816285,0.62066388577966,0.57974326262052,
0.50445557808449,0.62765221903449,0.58941974136783,
0.50310681110253,0.62777222285445,0.57752054473986,
0.50293040599919,0.63290237290405,0.59003651084035,
0.50019273296005,0.63042569584085,0.58628458445934,
0.50053787644289,0.63416479513706,0.5837026744232,
0.49821129620541,0.63122150119849,0.58582717613659,
0.498115808068 2,0.63293195384572,0.57471339800011,
0.49737736510679,0.63602291116938,0.5819882987768,
0.4954612960558,0.63679073183044,0.57716652447137,
```

Figure 11 – A valid configuration file. The first line showing the parameters and their values. All other lines contain the data set results. Contains 24 epochs with three data sets.

The user is allowed to upload files without any results. If there are some corrupt entries and a result value is not a number, the user will be informed via a toast message that the results may be incorrect.

4.5.2 ConfigSet Table

Every project's `ConfigSet` is displayed in the table at the center of the project page. The entries of the table can be filtered, which is explained in section 4.5.6, or personalized.

Entries in the table can be highlighted by clicking on them. Another click will remove the highlight effect. A double click on a `ConfigSet` will direct the user to the `ConfgiSet` page.

To adjust the table, the user can click on the settings symbol at the upper right corner of it. The shown modal holds options for the tables columns as well as the pagination settings (see Figure 12).

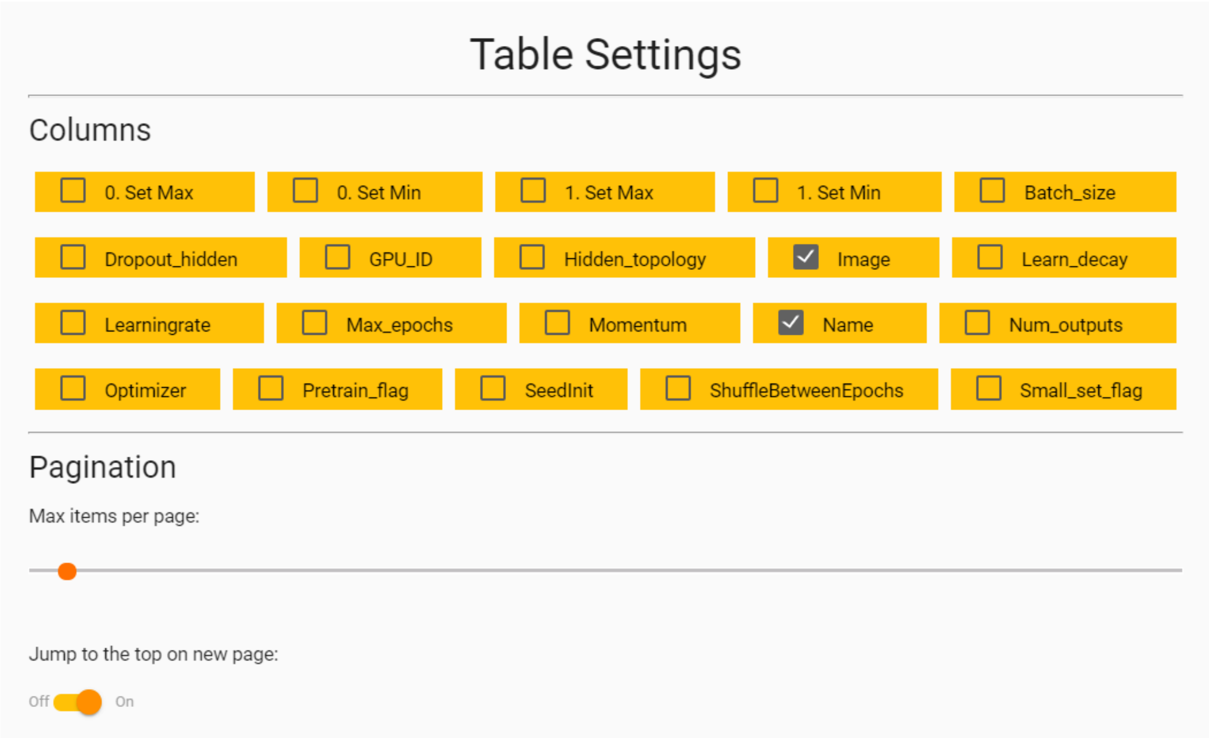

Figure 12 – ConfigSet table settings with column picker and pagination settings. This modal opens when the user wants to configure the ConfigSet table and clicks on the cogs button of the table.

Every column, except for the 'Actions' column can be selected or deselected. The number of shown entries per page can be adjusted, as well as the option to jump to the top of the page, when opening a new one. The table settings are saved with the user's profile for every project. There are some additional columns to choose, which are no parameters. For every found data set, the minimum and the maximum value of the results can be shown as a column as well as a thumbnail of the result diagram.

4.5.3 ConfigSet Page

On a `ConfigSet` page (see Figure 13) unnecessary parameters can be deleted and the results can be seen and extracted as a Scalable Vector Graphic (SVG). The page also offers the features to upload any related file, and edit the name or the description of the `ConfigSet`.

If the application found valid results in the uploaded configuration file, they will be displayed as a multiline chart at the top of the page. The y axis represents the error messure and the x axis the epoch. Every training set is shown as a line in a different color. When the user hovers over a data set value in the diagram, the matching result value will be shown above the mouse.

The chart diagram can be downloaded as an SVG file by first clicking on the convert button at the top left of the page and then clicking the appearing download button.

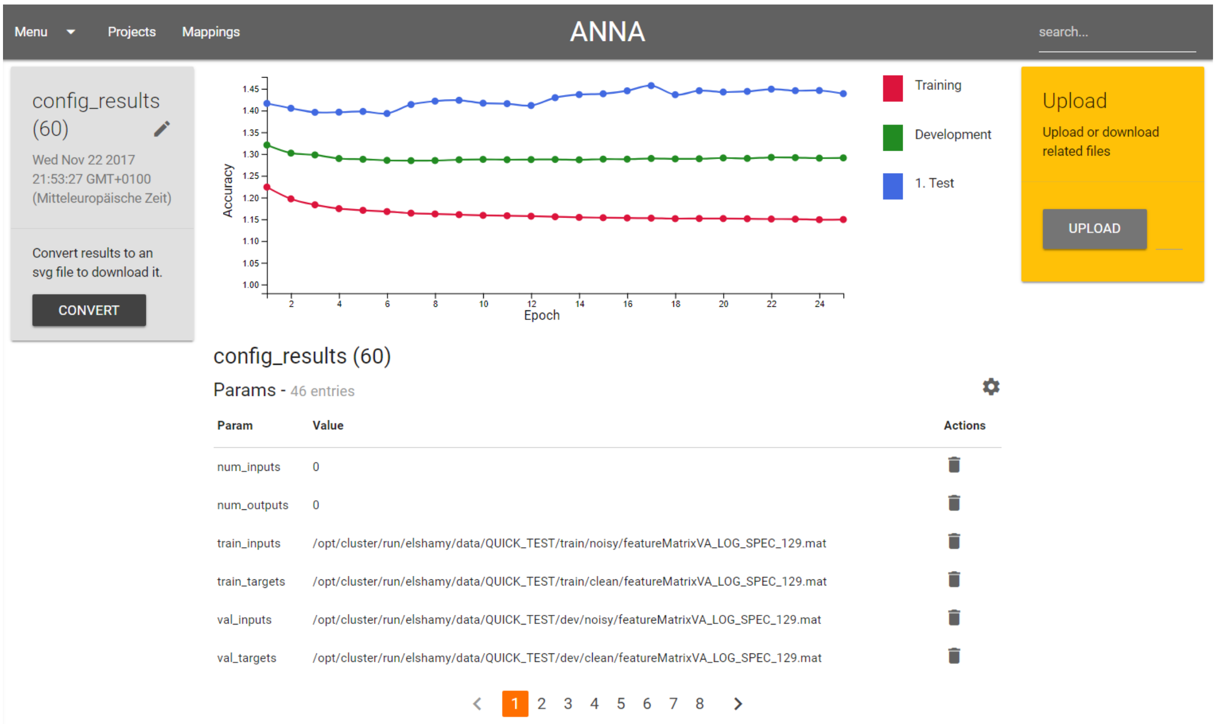

Figure 13 – A ConfigSet page, containing the description card, the results presented in a multiline chart, the parameters with their values and the button to extract the chart and download it. Additional files can be seen and uploaded on the right side.

Every found parameter-value pair is shown at the bottom of the page. The table can be sorted by parameter or value by clicking on the column header. Each parameter-value pair that is not necessary can be deleted by clicking on the delete button if the current user is creator of the `ConfigSet`. After confirmation from the user, the action will be executed. The parameter-value table can be filtered by using the search field of the top main bar.

The `ConfigSet` page offers the option to upload files, that are specific for this parameters and results. The files can be uploaded by the `ConfigSet` owner by using the 'Upload' button on the 'Upload' card. After at least one file is attached to that `ConfigSet`, the list of available files appears above the 'Upload' card. With a click on a file, it will be downloaded.

When clicking on the trash can symbol, the file will be deleted from the database after a confirmation from the user. This can only be done by the owner of the `ConfigSet`.

4.5.4 Mappings

One of the main features of the application is to filter between different `ConfigSets` in a project, to compare the accuracy, or the loss. In order to filter those, all `ConfigSets` within a project must fit the same criteria. A `Mapping` will accomplish that by assigning aliases to parameters, so that both are equivalently treated in the application. The relation is always between a parameter key, which behaves as a primary shown parameter, and several aliases.

A `Mapping` can be assigned to a project. If the current project has no `Mapping` assigned yet, it can be created by clicking on the ‘create new’ button of the `Mapping` card at the project page (Figure 10), but only if there is at least one configuration file uploaded. The first found `ConfigSet` will be used a base. All of its parameters become the key words in the new `Mapping`. All of the other parameters from other `ConfigSets` of the project will either be assigned automatically, if they are already a keyword or an alias or added as unrelated parameters, if they don’t match any existing. When creating or updating a `Mapping`, the user will be informed via a toast message, how many unrelated parameters the `Mapping` has.

Another option to relate a `Mapping` to the project is to select one of the existing ones at the select field of the `Mapping` card. After the user confirmed his action, all `ConfigSets` of the project will be searched for new parameters, which aren’t available in the `Mapping` yet.

If a `Mapping` is assigned to the project, it can be either updated or the relation between the project and the `Mapping` can be removed, by using the respective buttons on the card. Clicking the ‘remove’ button will clear the relation between the project and the `Mapping`, after the user’s confirmation. This will also remove the option to filter the `ConfigSets`. Every `Mapping’s` parameter key or alias that was added with this project will remain. Afterwards another `Mapping` can be assigned to the project. The ‘update’ button will add every new `ConfigSet’s` parameters to the unrelated parameters of the `Mapping`.

There are two different ways of getting to a `Mapping’s` page to assign aliases and set `Flags`. The first one is by clicking on the `Mapping’s` name on the card at the project page. The second one is by clicking on the `Mappings` link at the menu bar and afterwards choosing the right one.

A `Mapping’s` page (see Figure 14) contains all parameters and aliases as well as `Flags`, which are explained later (see section 4.5.5).

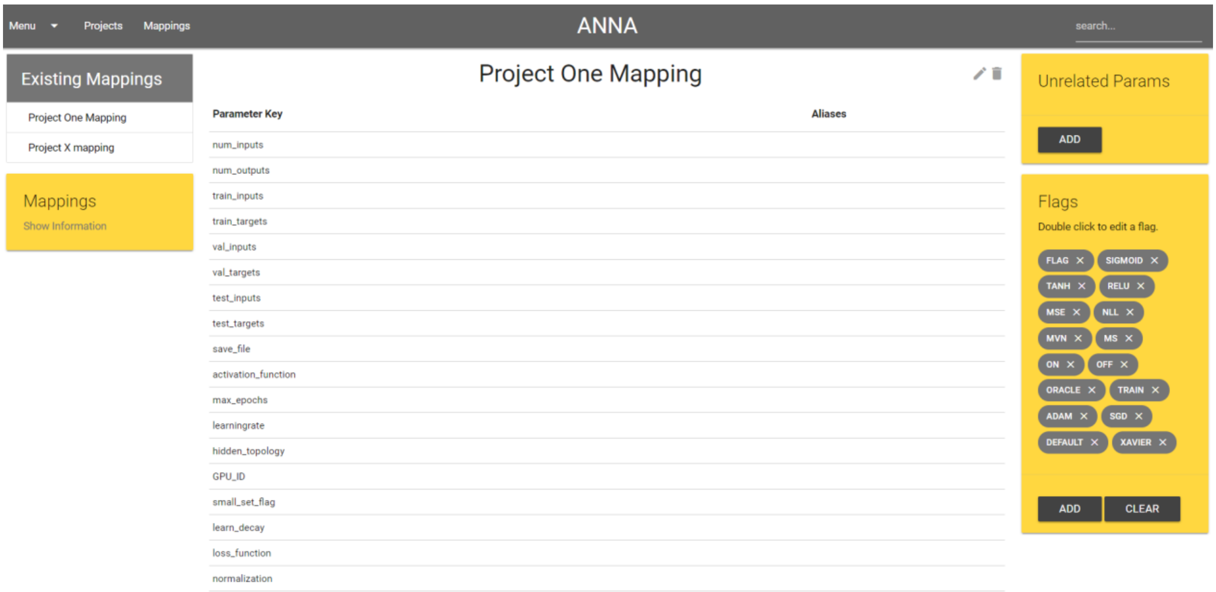

Figure 14 – A Mapping page without unrelated parameters. The left showing the existing Mappings and a short description on how to use Mappings. The middle area contains the parameters with their aliases and the left shows the unrelated parameters and the flags assigned to the currently chosen Mapping.

At the right of the page all owned existing `Mappings` can be selected to be viewed, and an informational text is displayed on how to use `Mappings`. Again, only the creator of the `Mapping` can edit it. The center and the right area are specific for the chosen `Mapping` and show their parameters with aliases and the `Mapping’s Flags`.

Here, the name of the `Mapping` can be edited by clicking on the edit button at the top of the page. Performing this action, will open a known edit modal. Next to the edit button there is the button to delete the currently selected `Mapping`. All projects, which are related to that mapping will lose their ability to filter the `ConfigSets` until they another `Mapping` is assigned to them.

To define an unrelated parameter to a parameter key as an alias, the user has to drag the chosen parameter and drop it onto the right row at the alias column. This way, the parameter is not unrelated anymore and will now have the same meaning as the parameter key of the destination row. He is equivalent to all other aliases in that row. This means, that the values for any parameter or alias in a row should also be valid for any other alias.

If an unrelated parameter does not match a key, it can become a new key when dragging it onto the add button of the ‘unrelated parameters’ card (at the upper right corner of a `Map–`

ping's page, Figure 14). Aliases can be dragged back to this card to unassign them. The aliases are not fix and can be adjusted at any time. Even dragging an alias to another parameter key is possible.

4.5.5 Flags

Flags are used to give parameter values a speaking description. A Flag holds a key and the corresponding meaning. Every found occurrence of the key will be translated to the description, if the corresponding project is assigned to the referring Mapping.

Flags can be created by clicking on the add button on the 'flag' card or by dropping a plain text file, containing Flags, on this card. When manually adding a new Flag, a modal opens, where the user has to type in the key and meaning (see Figure 15). After clicking the create button, the new Flag will be added to the Mapping.

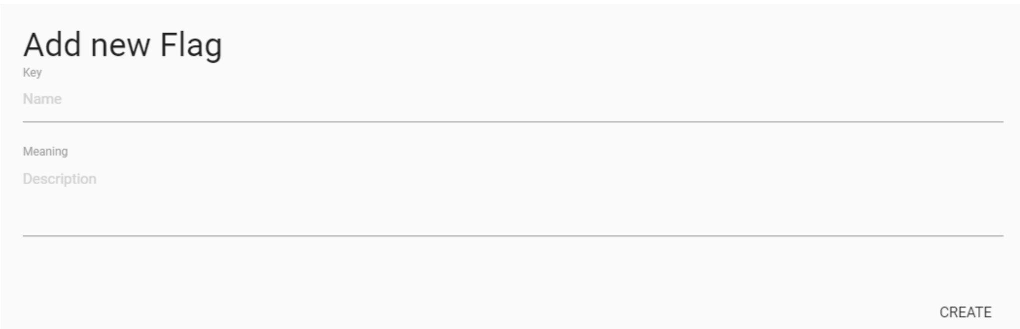

Figure 15 – Modal for manually creating a new flag, setting key and meaning.

To upload an existing Flag file, the user has to drop it on the 'flag' card. The file can contain as many Flags as needed. Every flag must have a single line and match one of the following patterns: `flag.meaning = key` or `flag.meaning : key`. The case sensitivity is not important here. A valid Flag file could look like shown in Figure 16.

```
FLAG.ON = 8
FLAG.OFF = 9
FLAG.GPU_ID_AUTO = 10
FLAG.GPU_ID_0 = 11
FLAG.GPU_ID_1 = 12
FLAG.GPU_ID_2 = 13
FLAG.GPU_ID_3 = 14
FLAG.NORMALIZE_ORACLE = 15
FLAG.NORMALIZE_TRAIN = 16
```

Figure 16 – A valid flag file. Assigning the meaning to a key.

`Flags` can be deleted by clicking on the cross at the flag itself on the ‘flag’ card. It is recommended to use hexadecimal representation to avoid collision with numerical parameters, e.g. the number of epochs. At the bottom of the ‘flag’ card all `Flags` can be deleted, by clicking the ‘Clear’ button. After a confirmation all flags for the chosen mappings get deleted. Again, `Flags` are only a visible component and do not influence filtering or other actions.

4.5.6 Filtering

One of the key features of the application is to filter `ConfigSets` by their parameters or results. This can only be achieved if the `ConfigSets` are comparable via a `Mapping`. Once a `Mapping` was assigned to a project, all project’s `ConfigSets` are filterable. This can be done on a project’s page (see Figure 10). To add a filter the user has to click on the ‘plus’ in the upper left corner of the page. The opened modal shows every found possible filters, containing parameters (see Figure 17).

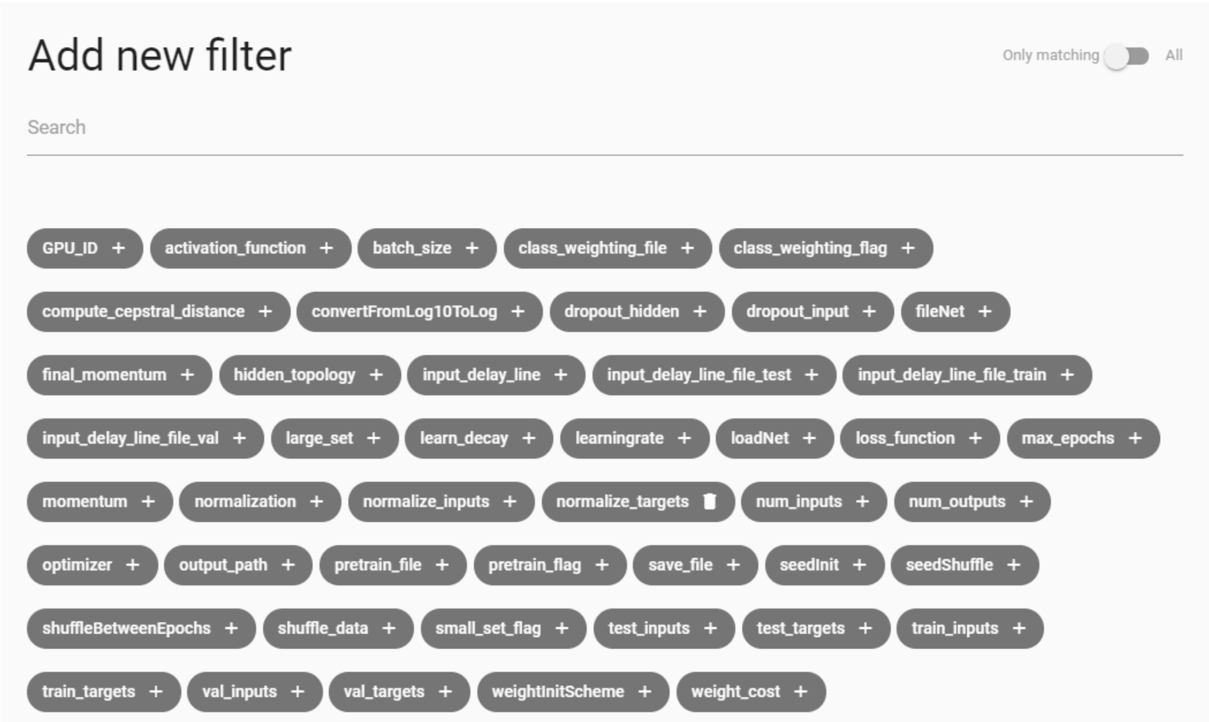

Figure 17 – The modal to add filters. Active filters show a trash can symbol (like normalize_targets), non-active show a plus. When clicking on a name, the active state toggles.

Activating the switch at the upper right corner of the modal will toggle between those filters, which are relevant to the project and every for the chosen `Mapping` found filter.

To add a filter, the user has to click once on the name or the plus behind the name. Already chosen filters can be identified by the trash can to the left of its name (like ‘normalize_targets’ in Figure 17). These can be deleted by again clicking on the name or the trash can. The top search bar of the modal helps searching and adding filters more easily. While typing, the list of possible search results will be displayed, and the user can select or deselect the filter he wants to. In order to close the modal, a left click on the blurred background is enough.

After a new filter was added to the project, a card with the name of the new filter is added to the left of the project page, but the list of shown `ConfigSets` remains the same. Every new filter contains all possible options for its kind. Not needed options should be deselected to cause an effect on the `ConfigSet` list.

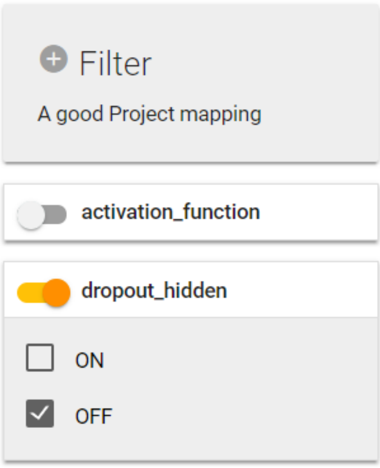

Figure 18 – The filter section with an active and an inactive filter. The activation_function filter is not active. The dropout_hidden filter is active and has only the 'OFF' option enabled.

Filters can be deselected in the modal, but deactivated or adjusted on their own card. To deactivate a filter, the user has to click on the switch of the filter card. Active filters have an orange switch, others a grey one. When clicking on a filter card, it opens and shows all the options for this kind of filter. The options are all available values for this parameter or its aliases in every `ConfigSet`. By default, all options are selected. An option can be deselected by clicking on the name or the checkbox. The `ConfigSet` list updates whenever a filter is changed. Those `ConfigSets` which doesn't match the filters will be hidden. Figure 18 shows the filter section with an inactive filter ('activation_function') and an active filter ('dropout_hidden') with one option enabled.

If an option is deselected, the `ConfigSet` list will only contain the `ConfigSet` that do not have this option as a value for the referring parameter. This behaviour includes that, if a `ConfigSet` does not have a matching parameter for the filter, it will not be affected by the filter. This means that even if all options of a filter are deselected, the `ConfigSet` list could still contain entries.

When deactivating a whole filter, it is still visible as a filter card, but does not affect the list of `ConfigSets`.

Every filter for every project is saved in the user's profile. Whenever the user visits a project page again, his old filters are still applied and visible.

5 Developers Perspective

This chapter should be used as a guide on how to further develop the application. At first some useful tools are introduced and explained. They enhance the developers coding experience and are valuable for debugging. Furthermore, the setup of the development environment is described. After that, we give a quick overview of the current project structure. Later on, a brief guide for coding is given, concerning new components, collections, working with `Observables` and extending other functionality.

To continue the development of this application, the developer should know a bit about Angular and Typescript. This chapter covers the basic concepts and teaches how to begin coding and enhancing the application. Not all structures and coding concepts can and should be handled here. There are good tutorials on the Angular [**Ang17**] and the Meteor [**Met17**] website, going more into detail. Because Meteor-specific code is more straight forward, we will not be talking about it much.

5.1 Developer Tools

At first there is to say, that every developer has its own preferences on programs for web development. In this section a recommendation for those programs is given.

The first two tools to run the application on a local machine are NodeJS [**Nod17**] and Meteor [**Met17**], which can both be downloaded at their own websites for free.

To edit, add or delete code an IDE is useful. The recommended program for that is Webstorm [**WeS17**], because it is clear and easy to use, has good code completion, even for Angular and Meteor, and is well maintained. A free alternative is Atom [**ATM17**] which can be very mighty with the proper extensions.

Another helpful tool is Robo3T [**R3T17**] for viewing and editing the database, which isn't necessary for development.

Lastly a preferred browser is needed for viewing and debugging the application.

5.2 Setting up a Development Environment

In this section the process of setting up the development environment on a developer's computer is explained. This includes, how to configure the needed programs and how to execute the source code.

5.2.1 Running the Application

If the source code is available on the local machine, the application will be almost runnable.

In order to run the application, all NodeJS packages have to be installed first. This can be done in a terminal or console. The developer has to navigate into the applications main folder, type

```
npm install
```

and hit enter. NodeJS should now install all the packages listed in the *package.json* file.

With that done, the application is runnable. The developer has to type in

```
meteor --settings settings.json
```

in the main folder and hit enter. The `meteor` command will launch a meteor application and the settings argument tells the application where to find the settings (explained in section 5.3.2). Now, all missing meteor packages, which are listed in the *packages.json* file in the `.meteor` folder, will be installed. This might take a few minutes. The local proxy, database, and server should start after that and display the address where to find the application. The application can be seen by navigating to that address in the browser (usually http://localhost:3000).

5.2.2 Setting up the IDE

A good IDE is essential for development. It should at least support HTML, JS, and CSS with code completion and syntax highlighting. The complete source main folder can be chosen as a project root and opened in the preferred IDE.

Optionally a run configuration can be set, to easily start the application from the workspace. The developer has to set `meteor` as a command with the argument

`--setting settings.json`.

The meteor executable can be found in

`%LOCALAPPDATA%/.meteor/meteor.bat`.

5.3 Overview

This section gives an overview of the folder structure and the settings file of Meteor, which allows to pass in data to the application before starting it.

5.3.1 Folder Structure

There are three main folders in the project root. The *server* folder contains code, that only the server can execute and see. The *client* folder contains the client side code, which the server sends to every client instance, and contains the Angular specific code. The third one, the *both* folder contains code, that the server as well as the client can see and work with. It holds models and collections.

The *client* folder contains the index file, which is used as a loading screen, before the application is fully loaded. It also contains the *imports* folder, which is the place, where all modules, helpers and services exist. The main page/component, showing the menu bar, is called `app.component` and owns an HTML, a Typescript, and an SCSS file.

5.3.2 Settings File

The settings file is a way to pass variables to client and server when starting the application. It is in JSON format and contains a key namend *public*, which can be accessed everywhere. All other keys can be accessed only from the server side.

If the settings file has been provided as a starting argument, then `Meteor.settings` and `Meteor.settings.public` are available.

In our case, the LDAP address and the distinguished names (DN) for the users at that LDAP system are stored in the settings file. When connecting to the LDAP to log in, both variables are used.

Those settings variables can be created at own discretion.

5.4 Angular Basics and Adding Components

Adding components is an easy and fast way to extend the application. Like mentioned before, a component can be any visual part with background functionality. For example, a component can be a modal, a whole page or even the `ConfigSet` filter, which already exists in our application. We will start by explaining how to add a new component and afterwards introduce some of Angular's features with an example of a component.

5.4.1 Creating and Declaring a Component

At first, a new folder with the name of the component should be created in the *client* folder. The developer has to add an empty `.ts`, `.html`, and `.scss` file to the folder. The component specific files should be named `name.component`, to distinct between components and other classes, like services. To implement a very basic component, only displaying a 'Hello World' text, the developer has to start by defining the component in the created Typescript file, as well as declaring it in the *app.module.ts*. A component is a simple class in Typescript, which has to be exported, so the whole application knows about it. For defining it as a component, Angular uses decorators over the class definitions. The basic `@Component` decorator should have a selector, a reference to the template (`.html` file) of the component and the corresponding style files. The selector will be the HTML tag to use this component. So, whenever the HTML tag appears in any template, this component will be placed there. This small component could look like shown in Figure 19.

```
import {Component} from "@angular/core";
import template from "./hello.component.html";
import style from "./hello.component.scss";

@Component({
    selector: 'helloWorld',
    template,
    styles: [style]
})
export class HelloComponent {

    constructor() {

    }

}
```

Figure 19 – hello.component.ts -A basic Typescript component class. On top the imports for Component, template and style. The Component decorator declaring the Component and the class itself with an empty constructor.

The *@Component* decorator can be used to define a lot more configurations, like animations, inputs or outputs from the parent component or even another change detection, that determines when the component should be rendered again.

After this is done, the component can be imported and declared at the *app.module.ts*. To do this, the developer simply has to put the class name of the new component in the declarations list and import the path, where to find the component.

The .html file could contain any HTML, without matching the known HTML structure. For example, just a text like ‘Hello World’ is fine. The .scss file should contain all the styles for this component to keep the overview.

With these few steps the new component can be used in every other component, by using the selector (<hello-world></hello-world>) in an .html file. The displayed component would just be a plain text saying, ‘Hello World’. The selector in the Typescript class is written in one word with the second word starting with a capital letter and can be freely chosen. The HTML template though uses this selector as shown with hyphen-separated words. This behavior is Angular specific.

This component can implement interfaces, inherit from other classes, have their own functions, use services and many things more.

5.4.2 Defining and Using Routes

If a component should become a new page with an own address, it needs a route. Routes are also defined in the *app.module.ts* file. A path and the component are needed for defining routes. Optional a Guard can be added to protect the view from specific user groups. For the example component a route looks like shown in Figure 20.

```
{
    path: 'helloWorld',
    component: HelloComponent,
    canActivate: [IsLoggedIn]
},
```

Figure 20 – An example route with Guard showing the path, the responsive Component and the Guard for the canActivate trigger.

The routes defined in the *app.module.ts* can be used in the router-outlet, which is the main part of the *app.component*, our main component. The HelloComponent is now reachable with the address ‘localhost:3000/helloWorld’ (in case the application is running on the local machine under the port 3000). The ‘canActivate’ property holds a type of Guard, that checks, whether the route can be accessed/activated or not. In our application, there is currently only the ‘IsLoggedIn’ Guard, which returns true if the user is logged in.

There are two common ways to use the defined routes. Angular offers the attribute ‘routerLink’ which can be attached to any HTML tag. This attribute needs a router path as an argument. When clicking on that HTML element, the route gets activated and the view switches.

The second way is by using code. The angular-router can be injected into a class by using Angular’s dependency injection and then used to navigate to the wanted view.

5.4.3 Angular Interpolation

Angular’s interpolation can be seen in the HTML templates whenever two curly brackets enclose an expression. The expression inside the brackets gets calculated and interpreted similar to Typescript. This is how we can use component variables from Typescript classes in the templates.

```
<p>The component variable 'name' says: {{name}}</p>
```

Figure 21 – A template with an interpolation of a component variable. The interpolation always shows the exact value of the variable.

The line shown in Figure 21 has a static text and the dynamic expression of the variable ‘name’ in curly brackets. This is an easy example on how to use Angular’s interpolation. Figure 22 shows the known ‘Hello World’ example component with the simple string variable called ‘name’. To show this variable in the referred template, Angular’s interpolation is the best way to do so. Whenever this variable changes, the shown expression will also change.

```
import ...

@Component({
    selector: 'helloWorld',
    template,
    styles: [style]
})
export class HelloComponent {
    public name : string = "Hi!";
}
```

Figure 22 – The HelloWorld component with a public string variable.

This is not the only thing that interpolation can do. It behaves a lot like Typescript code. Inside the curly brackets, component methods can be called, or mathematical expressions can be calculated. However, using it for HTML text output is nice, but it can also be used for attributes or properties of HTML tags, like the source URL of an image or the ID attribute of a tag.

However, as the values of interpolation expressions are newly calculated after every change detection cycle of Angular, only quickly executable expressions should be used.

5.4.4 Data Binding in Angular

Angular provides easy-to-use functionality like mapping HTML events to the functions in a component or linking a component variable to an HTML text, like we showed in the previous section. The technical term for mappings or links is data binding. Angular can have three forms of data binding.

One-way binding from a data source to the view target is what we saw in the interpolation section. There is also a second way of using this one-way binding: with the square bracket notation.

Square brackets also declare data-binding from the component to the template. This can be used for example to bind an *innerHTML* attribute of a headline to a variable in the component. Whenever the variable changes, the headline text changes to that variable's value without any delay. For this example, the same effect can be achieved through Angular's interpolation syntax, which are the two curly brackets around the expression. Figure 23 shows the two different approaches, which lead to the same result.

```
<!-- Square bracket data binding -->
<h1 [innerHtml]="selectedMapping.name"></h1>

<!-- Interpolation binding syntax -->
<h1>{{selectedMapping.name}}</h1>
```

Figure 23 - A component variable expressed in two ways. The first one showing the binding of the 'innerHtml' attribute to the variable, the second one showing the string interpolation binding.

The second way of data binding Angular uses is the one-way data binding from the view target to the data source. This is useful, if the component should react on an HTML event, or a component variable should be bound to the value of a select tag.

For example, to link a button click to the function 'button1clicked()' of a component, an attribute '(click)' is added to the button, which holds the function as an argument. This is shown in Figure 24.

```
<button id="button1" (click)="button1clicked()"></button>
```

Figure 24 – HTML tag for a button with the ID ‘button1’ and the Angular event ‘(click)’ pointing at the components function ‘button1clicked()’ that will trigger, whenever the button was clicked.

Thus, when the HTML event ‘click’ is triggered, it calls the given method in the component.

The third and last way of Angular’s data binding is the two-way data binding, which combines the two previous ones. To use the two-way data binding, the round and the square brackets are needed around the target. This is extremely helpful for input forms like a text line. If a value for the bound variable is predefined by the component, the text line will show the value when entering the view and when the user edits the text line, the component variable will be adjusted without any delay.

5.4.5 Directives

Angular also offers many predefined, so called *directives*, which are used like an HTML selector or an HTML attribute. The most useful and most frequently occurring directives are `*ngFor` and `*ngIf`. Both can be used on any HTML tag as an attribute.

The `*ngFor` directive can iterate through an array-like object like a for-loop. If the component owns an array variable, containing names, listing all items can be done like shown in Figure 25. Whenever the ‘names’ variable changes, the displayed list will also change immediately.

```
<div *ngFor="let name of names">{{name}}</div>
```

Figure 25 – An example of the *ngFor directive. Displaying all names of the names array.

For dynamically hiding or showing template parts the `*ngIf` directive is very useful. Like the `*ngFor` directive it can be used as an HTML attribute and hold an expression as an argument. Whenever the expression is true, the tag with the directive will be visible, otherwise it will not.

5.4.6 Pipes

Angular offers a great way to filter and manipulate arrays before they are displayed. The constructs needed for that are called pipes. A pipe must be implemented in a Typescript class and can then be used in template files. In our application there are a few pipes used for sorting arrays, or filtering for the owned projects and `ConfigSets`. These existing filters are stored in the file *filter.pipe.ts*. All pipes must implement the `PipeTransform` interface to implement the `transform` method.

A helpful filter is used in Figure 26. It filters the given array and returns the items that match the given search text, which is put in as a parameter after the pipes keyword.

```
<div *ngFor="let name of names | filter: 'Peter'">{{name}}</div>
```

Figure 26 – Usage of a filter pipe. Iterates through the names array, which is filtered by the filter pipe. Only the names are displayed, that contain the string 'Peter'.

Figure 27 shows the implementation of that filter. It gets the input array of strings and the search text as parameters.

```
/**
 * This filter pipe is used to filter strings by the given searchText
 */
@Pipe({
    name: 'filter'
})
export class FilterPipe implements PipeTransform {
    transform(items: string[], searchText: string): string[] {
        if(!items) return [];
        if(!searchText) return items;
        searchText = searchText.toLowerCase();
        return items.filter( callbackfn: it => {
            return it.toLowerCase().includes(searchText);
        });
    }
}
```

Figure 27 – An Angular pipe for filtering a string array by a search text. The result will be all input strings, that contain the given text.

After checking if the input or the search text is existent, it returns only the input items, that contain the search text. The implemented filter has to be declared in the *app.module.ts* in order to use it in any modules `.html` file. The class decorator is another Angular decorator. It declares the following class to be a pipe and sets the name to use this filter in a template.

When filtering the items in the `transform` function, they can also be modified in any way. For example, changing every letter to a capital one or adding another property to an object.

5.5 Adding and Extending Collections

Collections store data of one entity. The application has three self-made collections and a collection given by Meteor, the `user-collection`. The application specific collections are the `ConfigSets` collection, the `projects-collection`, and the `Mappings-`

`collection`. If a new entity should be added or an existing one needs to be edited, this guide will help, where to start and what to do.

A collection needs two files, an interface or a class determining what kind of objects are stored in the collection and the collection export itself, connecting to the database. These files need to be stored in the *both* folder, so client and server know about them. The interface or the class can be stored under the *models* folder, next to all existing model definitions. The definition of the collections object can be a simple class which then is exported, so it can be used in the whole application. A typical example of such a class is shown in Figure 28.

```
export class Project {
    name: string;
    description: string;
    mappingID: string;
    creator: string;
    _id: string;

    constructor(
        name: string = '', description : string = '', creator: string = "", mappingID: string = ''
    ) {
        this.name = name;
        this.description = description;
        this.mappingID = mappingID;
        this.creator = creator;
    }
}
```

Figure 28 – A typical collection's object class definition. A typescript class containing the attributes and a basic constructor.

The constructor is not needed for collection purposes, but for later work with the class. In fact, just the variable declarations are enough. This class can then be used to tell a collection, that this is the type of objects it should store. If an existing collection shall be edited or adjusted, this file can be easily modified to add a parameter for example.

With the existing object definition, the collection itself can be created. Figure 29 shows the entire *project.collection.ts* file, which essentially is a complete declaration and definition of a MongoDB-collection.

```
import { MongoObservable } from "meteor-rxjs";
import { Project} from "../models/project.model";

export const ProjectsCollection = new MongoObservable.Collection<Project>( nameOrExisting: "projects-collection");
```

Figure 29 – Definition of a collection. Importing the MongoObservable and the collection's object class definition and then creating the new MongoObservable.

The project class is imported, to declare the type for the stored objects, and a `MongoObservable` is imported, too. By creating a new collection from this `MongoObservable` with a generic type (`Project`) and the name of the collection on database side (`projects-collection`) Meteor creates a new collection, which then is exported and can be used in the whole application. A database service should handle the data transfer from this point. Data services and `Observables` are further explained in the following sections.

5.5.1 Storing Files with GridFS

We also use a MongoDB specification called GridFS, which lets us store any files, including binaries in the database. Because MongoDB allows only JSON conform data up to a size of 16MB, we need this specification to provide a storage location for any related files. GridFS splits the files into smaller chunks and stores a meta file, that contains information about the chunks. Via this meta file, the stored files can be reassembled and reached via a URL. This system has the advantage of a portable data pool, that only needs the database files, not any others that are stored at the file system itself.

To use GridFS properly with our known concepts, we used a Meteor specific wrapper.

5.6 Data Services and Observables

All data actions for our application are handled in the data service, which is responsible for the corresponding collection. When getting data from the database, it takes a bit of time to fetch all results and prepare them for the client, in order to show them. Angular and another package called RXJS offer an easy way to deal with those tasks, which are called `Observables`. The data services we implemented are so called `Injectables`, which means that they have one instance and can be injected where they are needed. `Injectables` come natively from Angular's dependency injection, which we mentioned earlier.

5.6.1 Observables

An `Observable` is used for handling an asynchronous data stream, when data is fetched in sequences over time. Any observer can subscribe to that `Observable` and gets notified whenever a new data sequence is available. Well maintained documentation of `Observables` can be found at the RXJS website [**RJS17**].

MongoDB and the server can't handle every request the client is sending instantly. Instead of active waiting, notifying the client when the data is ready is better. In our application this is done by RXJS' `Observables`.

```
1  public createObservable() : Observable<number> {
2          return Observable.create(observer => {
3              setTimeout(() => {
4                  observer.next(42);
5              }, 1000);
6  
7              setTimeout(() => {
8                  observer.next(43);
9              }, 2000);
10 
11             setTimeout(() => {
12                 observer.complete();
13             }, 3000);
14         });
15     }
```

Figure 30 – A simple observable returning two numbers within two seconds and completing after three seconds.

Figure 30 shows a function returning a simple `Observable`, which delivers the type `number` as data. The type of data the `Observable` returns is set automatically. It can also return many different types. This function however asks for a ‘number’ `Observable`, which can be seen by the generic type tags in line 1 of Figure 30. The `Observable` creates one return value, ‘42’ after one second, and the next one, ‘43’, after two seconds. In this example this is done by using a typical JS `setTimeout` function, to simulate a delay. In the real application this could be the time the MongoDB needs for finding the results. An `Observable` can complete or stay open, depending on the way it is used. In our example the `Observable` completes after three seconds, meaning that the data stream has ended and the correspondence between observer and `Observable` is finished. Any class that knows the `Observable` can be an observer, which is shown in Figure 31.

```
1  private observable : Observable<number> = createObservable();
2  
3  this.observable.subscribe((data : number) => {
4      console.log(data);
5  }, () => {
6      console.log('Something went wrong')
7  },() => {
8      console.log('Completed!');
9  });
```

Figure 31 – A basic observer subscription with ‘next’, ‘error’, and ‘completed’ function. The ‘next’ function prints the incoming data to the console.

The first line shows, that the `Observable` is known and can be used. Oftentimes only a reference to an original `Observable` is needed, this is called `ObservableCursor`, which works the same way. When subscribing to an `Observable`, there can be three different events to listen to. All three events give input and can be implemented using a self-made function.

The first one called `onNext` triggers whenever the `Observable` sends a new data sequence. This data sequence will be the input for the function. In our example we called the input ‘data’. Because the `Observable` returns only numbers, the ‘data’ input must also be a number. For keeping the overview, this can be made clear by putting the type after the argument, divided by a colon. Whenever this colon appears in Typescript, it means a type is being declared. In the example the data is just printed to the console – ‘42’ after one second and ‘43’ after two seconds.

The second event to listen to is the `onError` event, which is called whenever an error appears, and the `Observable` does not function properly. This means that no other event will be submitted after this one. The `onError` event takes as its parameter an indication of what caused the error.

The `onCompleted` function will be called if the `Observable` calls `complete()` after the last `onNext` was called. It has no parameter. This is the place where to clean up after the final response. In the example a simple ‘Completed’ is printed at the console after three seconds.

This is only the basic functionality of an `Observable` and there are also a lot more concepts from RXJS, like ‘Subjects’, which can function as an observer and an `Observable` combined, listening to streams and emitting data.

5.6.2 Data Services and MongoDB Queries

In our application every collection has a data service to handle and distribute the data, mostly implemented using `Observables`. Those data services should provide all functionality the client needs when fetching, editing, creating or deleting the database entries. This means, that these classes are the only way to manipulate the data. Because the MongoDB delivers `Observables` or `ObservableCursors` when the server requests something, our data services will be based around those constructs.

One simple example of how the data queries work is presented in Figure 32.

```
public static getConfigById(ID): ObservableCursor<ConfigSet> {
    return ConfigSetsCollection.find( selector: {_id: ID});
}
```

Figure 32 – A simple data query to get a configuration set by its ID. The selector matches a collection attribute to the given ID value.

This method is used for example, when the user enters a `ConfigSet` page (see Figure 13). The application requests the `ConfigSet` with the corresponding ID, which is also part of the address to the page, via the `ConfigSet` data service. To find a single `ConfigSet` an ID as a parameter is all the function needs. It uses the collection method 'find' and a selector to fetch the matching data. A selector can be as simple as in this example, only comparing the property key from the collection with some value, but also be very complicated if the requested data is more specific. A complete guide on how to use data queries for MongoDB and the documentation can be found at www.mongodb.com [**Mon17**].

As previously mentioned almost all data service functions use `Observables` as a return value, so handling the actual data happens in the requesting class. Figure 33 shows a simple way of working with these return values.

```
this.route.params.subscribe( next: (params) => {
    this.id = params['id'];
    this.mappingDS.getMappingById(this.id).subscribe( next: (foundMappings) => {
        this.selectedMapping = foundMappings[0];
    });
});
```

Figure 33 – Resolving route parameters when the route.params Observable returns the parameters. Then calling the Mapping data service to get the reffering Mapping also with an Observable.

In this example a `Mapping` is fetched from the database using the `Mapping` data service (`mappingDS`). The mapping's ID is resolved from the route parameter called 'id'. Because the route parameters are not instantly available when switching a route, they are also delivered as an `Observable`, which can be subscribed to. After the parameter data sequence of that observable are available, the `Mapping` data service will be used to get a `Mapping` to the corresponding ID. Again, this will return an observable. If some `Mappings` are found, it will return them, and the component can finally save its data.

5.7 Extending Functionality with NodeJS

Extending the functionality can be done by implementing everything from scratch. But sometimes it is better to use existing code to save time and effort. A great way to use existing code is NodeJS and its package manager NPM.

Due to NodeJS and its big community there are thousands of packages, which are freely available to use. Whenever a complicated feature should be implemented, there might be a package for it to fit the purpose. Some of those are well implemented and maintained and some less. New packages should be handled carefully, because they could have dependencies with a large overhead or potentially be insecure.

Those packages can be found on the NPM website *www.npmjs.com* [**NPM17**] or at *www.atmospherejs.com* [**AJS17**] which holds a large set of meteor specific packages.

To install a package, their documentation page often shows a command. This mostly looks like

```
npm install 'package-name'
```

or

```
meteor add 'package-name'
```

for the meteor related packages. After the command was entered in a command prompt at the apps main folder, the referring package file will update and the package will install.

5.8 Documentation

The documentation of the application and the code itself is required to ensure a long-lasting tool, that is easy to maintain. This includes technical as well as functional parts of the application. The existing code is documented using comments. The functionality of the application itself is explained in the previous chapters. Whenever the application's functionality is further extended, the documentation should be updated in time, to keep the overview for future enhancements.

5.9 Deployment

Deploying the application to a server and updating it takes a few steps, which are explained in this section. We start by introducing the requirements a server needs to fulfil in order to be able to host the web application, continuing by how to initially setup the server, and in the end we will go through the updating process.

5.9.1 Requirements

The server can be either a Linux or a Windows machine. There are four tools to install, so the database runs and the server runs with a secure connection.

At first NodeJS has to be installed on the server (find the preferred installation on the NodeJS website [**Nod17**]). As mentioned earlier it is the platform Meteor builds on. With the installation of NodeJS, the NodeJS package manager will be installed. This means all the `npm` commands are available.

To store the data, a MongoDB instance is required. When setting up the developer's environment meteor automatically installs a MongoDB which is enough for those needs, but on the real server we need an independent installation of the database. Again, a matching installation can be chosen on the MongoDB website [**Mon17**], either for Linux or Windows. Any configuration of the database is not needed, we just need the address of it, so we can refer to that when setting up the server. Also, it is recommended to make the database only accessible from the local machine to prevent others from manipulating the data from outside our application. The database itself can be started by executing the *mongod.exe*, which is localized at `ProgramFiles/MongoDB/Server/version/bin/mongod.exe`. When starting the database, it usually uses the port 27017, which we will use later to configure our application start.

The next thing the application needs is a way to manage the running application and the Secure Socket Layer (SSL) connection. For this we chose the Passenger [**Pas18**] for management and NGINX [**Igo18**] as a reverse proxy server to enable a connection via Hypertext Transfer Protocol Secure (HTTPS). Passenger and NGINX can work together with a Passenger module.

At first a basic installation of NGINX has to be setup. This can either be done by downloading an installer on the website or by using commands, the according documentation can be found on the NGINX website.

Installing Passenger for the right environment is important. Fortunately, the website holds the documentation for installing Passenger on nearly every viable machine [**Phu18**]. With this step done, the NGINX module should be installed too.

Now, that all tools are installed the source code of the application needs to be transferred to the server. Meteor has a command to build the complete application, which includes compiling and minifying the code, so it is harder to read and is more bandwidth efficient. The command needs a location to build to, as well as an optional architecture argument, when the application should run on another architecture than this command is running at. Of course, the command should be executed in the main folder of the application. For our case the command looks like this:

```
meteor build ..\ --server-only --architecture os.linux.x86_64
```

These are all the requirements the application needs to run. Installing NGINX and Passenger is optional but very recommended because it makes maintaining the website much easier and without NGINX or a comparable tool, the application would not run via HTTPS. The use of SSL to provide a secure connection is almost required, when user specific data is transmitted between server and client. Sending data like passwords and usernames unencrypted is a lack of security.

5.9.2 Running the application

In order to run the application NGINX and Passenger have to be configured to listen to the database and knowing what application to start. The Passenger configuration file *passenger.conf* should include the following lines:

```
passenger_root /usr/share/ruby/vendor_ruby/phusion_passenger/locations.ini;

passenger_instance_registry_dir /var/run/passenger-instreg;
```

And the configuration for NGINX should look like shown in Figure 34.

```
1  server {
2      listen 134.169.30.190:444 http2 ssl;
3      server_name 134.169.30.190;
4
5      ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
6      ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;
7      ssl_dhparam /etc/ssl/certs/dhparam.pem;
8
9
10     # Tell Nginx and Passenger where your app's 'public' directory is
11     root /var/www/NNDO/bundle/public;
12     # Turn on Passenger
13     passenger_enabled on;
14     # Tell Passenger that your app is a Meteor app
15     passenger_app_type node;
16     passenger_startup_file main.js;
17
18     # Tell your app where MongoDB is
19     passenger_env_var MONGO_URL mongodb://localhost:27017/myappdb;
20     # Tell your app what its root URL is
21     passenger_env_var ROOT_URL https://134.169.30.190:444;
22     passenger_env_var METEOR_SETTINGS '{"public": {"ldap": { "url":
23     "ldap://ifn05.ifn.ing.tu-bs.de", "dn":
24     "CN=Users,DC=ifn,DC=ing,DC=tu-bs,DC=de"}}}';
25
26 }
```

Figure 34 – The NGNIX configuration file. Contains server name and port, SSL key paths, passenger configuration and environment variables for the application.

At first the server address and the name of the server are specified. To run the server with SSL, there are two keys and a parameter file needed. For our application those are created with a free-to-use software, but they could be purchased for having verified certificates.

The next few lines configure the passenger. At line 15 the app type is configured to NodeJS and the line after that specifies the startup file for our application, which is the *main.js*. For the use of the database, the connection URL has to be set as an environment variable, which includes the database type, the address with the port and the database instance to create. Next, the root URL of our application is again set as an environment variable. At last, our settings file, which is needed to pass in variables to Meteor before starting the app is put into one string and again set as an environment variable.

At this point, the application is built and stored at the server, NodeJS is installed, the passenger and NGINX configurations are done, and the database is running. The next step is to unzip the application bundle, install the node packages, give rights to the NGINX and start the application with passenger.

This can be done with a script, which we created. The script includes the following commands:

```
tar xzf BA.tar.gz

cd bundle/programs/server

npm install –production

npm prune --production

sudo rm -fr /var/www/NNDO/bundle

sudo mv /home/’username’/bundle /var/www/NNDO/.

sudo chown -R nginx:nginx /var/www/NNDO

sudo passenger-config restart-app /var/www/NNDO/bundle
```

Now the application is running locally at the port 3000 and can be reached at the address specified in the configuration. In our case the applications address is *https://134.169.30.190:444* because NGINX blocks the port 3000 from external accesses and redirects the port 444 with SSL to our application.

5.9.3 Updating the Application

Updating the application is an easy process. Because the setup and configuration are only initially needed, there are just two steps for the updating process.

At first a new bundle has to be created on the developer’s machine. This can be done, like shown before with the command

```
meteor build ..\ --server-only --architecture os.linux.x86_64.
```

After the bundle was transferred to the servers file system, running the script again will unzip the bundle, and restart the application. This should only take a few seconds downtime for the server, if properly executed.

6 Conclusion

With the work of this thesis we developed a web frontend to organize and manage deep learning research configuration files as well as results. We created an application capable of creating projects, storing configuration files with results, mapping parameter sets of different origins, and filtering the data. The tool is secured with authentication, using the institute's LDAP system. It uses modern web frameworks on the server and the client side that are well maintained and documented and therefore very long-lasting. In addition to that, the code behind the application is easy to understand, which makes further development possible.

The created application should be used as a base and can be enhanced in the future. It serves the basic needs to get a better overview of the work. Thanks to the easy extensibility of our chosen frameworks, extending the application with more features is supported. For example, the application could have access to the graphics processing unit cluster of the institute, so commands and programs could be started from the web frontend. This could include the automatic storage in the applications database, which means that the results and configuration files don't have to be saved on the local machine. Furthermore, meta studies of the stored data can be added to the application, analyzing the data on a higher level.

This bachelor thesis explains the architecture of the application by going into detail with the server side, the client side, the chosen database and the architecture composition. It shows the thoughts behind the software design concerning the programming language, some concepts of the frameworks used, the data handling, and the authentication system.

To give the application's users a good way of knowing and learning the functionality of the web frontend, the user's perspective is shown. This user perspective explains some technical terms of modern web pages, the basic structure the pages have, how the profile and login system work, what projects are and how they should be used, and how configuration files are managed.

For the developers, who want to enhance the created application a guide of development is given. The developer's perspective considers the right tool choice, how to set up the development environment, getting started, using the frameworks we chose and working with the database. At last, the deployment of the application to a server is explained.

A Bibliography

[Met17] © 2017 Meteor Development Group Inc. (2017, Nov.) Meteor.com. [Online]. https://www.meteor.com/

[Nod17] Node.js Foundation. (2017, Nov.) NodeJS. [Online]. https://nodejs.org/en/

[Ang17] Google Inc. (2017, Nov.) Angular. [Online]. https://angular.io/

[Mon17] MongoDB, Inc. (2017, Nov.) MongoDB. [Online]. https://www.mongodb.com/

[NPM17] npm, Inc. (2017, Dec.) NPM. [Online]. https://www.npmjs.com/

[Goo17] Google Inc. (2017, Nov.) Google. [Online]. www.google.de

[ECJ17] ECMA international. (2017, Dec.) json.org. [Online]. https://www.json.org/

[AJS17] Percolate Studio Ltd. (2018, Jan.) AtmosphereJS. [Online]. https://atmospherejs.com/

[ECM11] ECMA international, ISO/IEC 16262:2011, 2011-06, ISO/IEC 16262:2011 defines the ECMAScript scripting language.

[Typ17] Microsoft Corporation. (2017, Dec.) Typescript. [Online]. https://www.typescriptlang.org/

[CNE17] Natalie Weizenbaum, Chris Eppstein Hampton Catlin. (2017, Dec.) SASS. [Online]. http://sass-lang.com/

[RJS17] (2017, Nov.) ReactiveX. [Online]. http://reactivex.io/

[WeS17] JetBrains. (2017, Dec.) Webstorm. [Online]. https://www.jetbrains.com/webstorm/

[ATM17] Open Source. (2017, Dec.) Atom. [Online]. https://atom.io/

[R3T17] 3T Software Labs GmbH. (2017, Dec.) Robo3T. [Online]. https://robomongo.org/

[Pas18] Phusion Holding B.V. (2018, Feb.) Passenger. [Online]. https://www.phusionpassenger.com/

[Igo18] Igor Sysoev. (2018, Feb.) NGINX. [Online]. https://nginx.org/en/

[Phu18] Phusion Holding B.V. (2018, Feb.) Deploy Meteor with Passenger. [Online]. https://www.phusionpassenger.com/library/walkthroughs/deploy/meteor/

B List of Figures