Technische Universität Braunschweig

Institut für Nachrichtentechnik

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

Keywords:

Contents

1 Architecture

Since the basic idea of this tool is to give a lasting web frontend for the institute, a good choice of what software to use is essential. Therefore, for client and server-side code we chose well maintained and well-known frameworks as there are Meteor [**M1**] as a mostly server side JavaScript (JS) framework and Angular [**A1**] 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 [**Mo1**] as a database, which is explained later (see 1.3).

1.1 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 [**N1**] server. 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.

1.2 Client Side

On client side we chose the JS framework Angular in version 4, developed and maintained by Google Inc. [**G1**] . 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.

1.3 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 JSON-Object (JavaScript Object Notation [**ECM17**]) 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 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.

2.1 Typescript

Both, our client and our server almost fully consist of JS code. In fact, we use ECMAScript 6 (ES6) [**E1**], 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 [**T1**] 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.

2.2 Components

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

2.2.1 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 SASS [**SA1**], 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.

2.2.2 Routing

The *AppComponent* contains the headline navigation and a router outlet. This outlet is a place holder for any component, we want to navigate to in our application. A good way to deal with routing is the Angular basic package *angular-router*. We can easily define routes and their corresponding component and even add so called guards to grant access control over the routes. A route path is the name of the route you need when you want to navigate to the view as well as the additional part of the URL in the browser. For example, if the view shown is the dashboard, where you can manage and navigate to your projects, the route defines the path as '/dashboard' and the component to the *DashboardComponent.* To show the view, simply navigate to it with code or add the path to the 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 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.

2.3 Data-handling

There are two important parts to differentiate when talking about the data-handling in the project. The first part is the data stored in the database, which we extract using services. The second part is the design of the individual types and classes we created. When getting 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)* [**RX1**].

2.3.1 Data Model

Our chosen database, MongoDB, is a NoSQL. 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 straight forward code which is better to understand. In the following section we will explain how the data we need to persist in our application is stored.

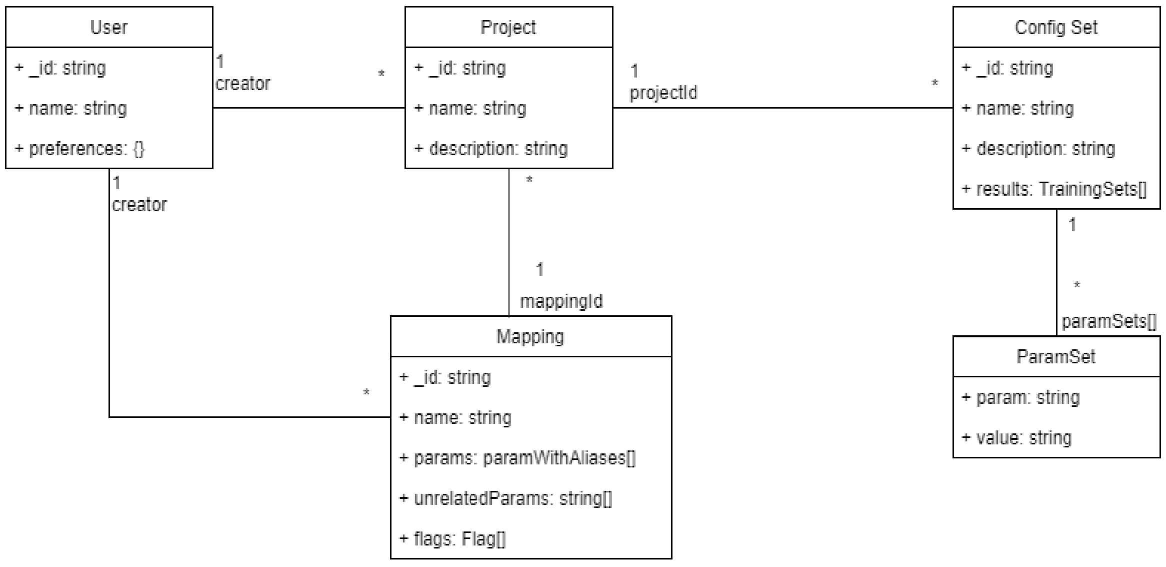


Figure 1 - Abstract data model

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

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

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

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

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

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

2.3.2 Data Services

For every collection in our database we have a data service in our application to control the data flow. A data service handles the queries and distributes the documents to the components. The most common queries are those to create a new document and update or delete an existing one. Every data service has a reference to the collection, for example the ConfigSet data service has a reference to the config-set-collection. The client as well as the server are aware of all collections. However, the queries are only made on client side.

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

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

2.3.3 Observables

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

2.4 Authentication

The whole application will be exclusively available for employees at the institute for Communications Technology or those who have an account at their 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.

3 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.

3.1 Technical Terms

3.1.1 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 2) or 'Successfully created'. These toasts last for a few seconds and can be dismissed by dragging them to the side.

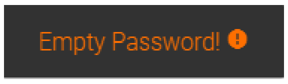
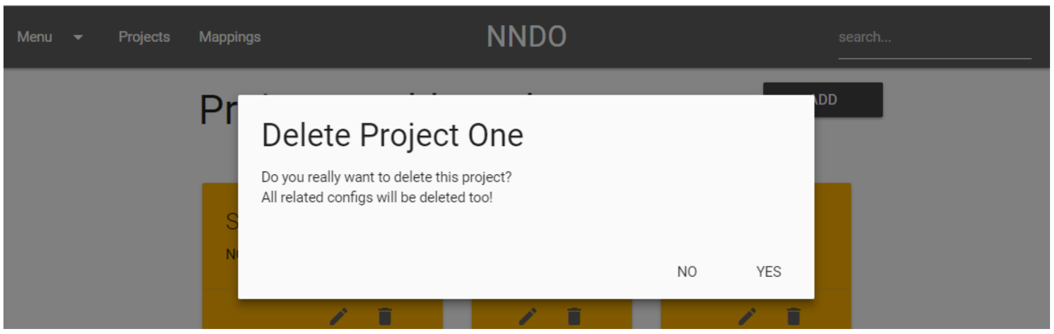

Empty Password!

Figure 2 – Example toast with error message

3.1.2 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 the 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 3), can't be dismissed. Those modals often contain two or more buttons, giving the user options to close the modal.



Delete Project One

Do you really want to delete this project?
All related configs will be deleted too!

Figure 3 – Confirmation modal for deleting a project

3.2 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 4).

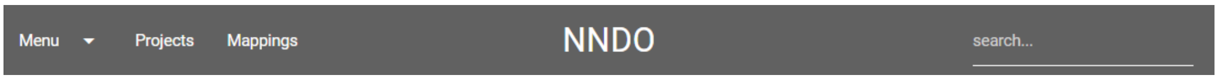


Figure 4 – The menu bar on a desktop browser

It is a fixed bar containing links to the project dashboard and the mappings page as well as a dropdown menu for actions like the 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.

3.3 Profile/Login

Like explained earlier, the application is only accessible for those, who have an account at the institutes LDAP system. In order to use any functions 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 (see Figure 5).

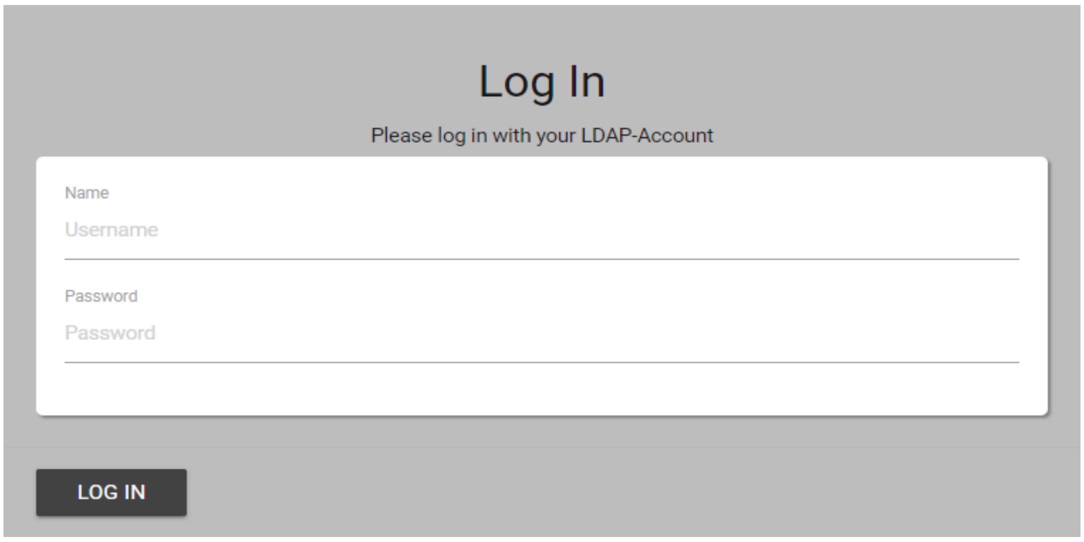


Figure 5 – The log in page with username and password form and log in button

Here he can type in his username and password. After clicking on the log in button, the user gets notified about the success 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.

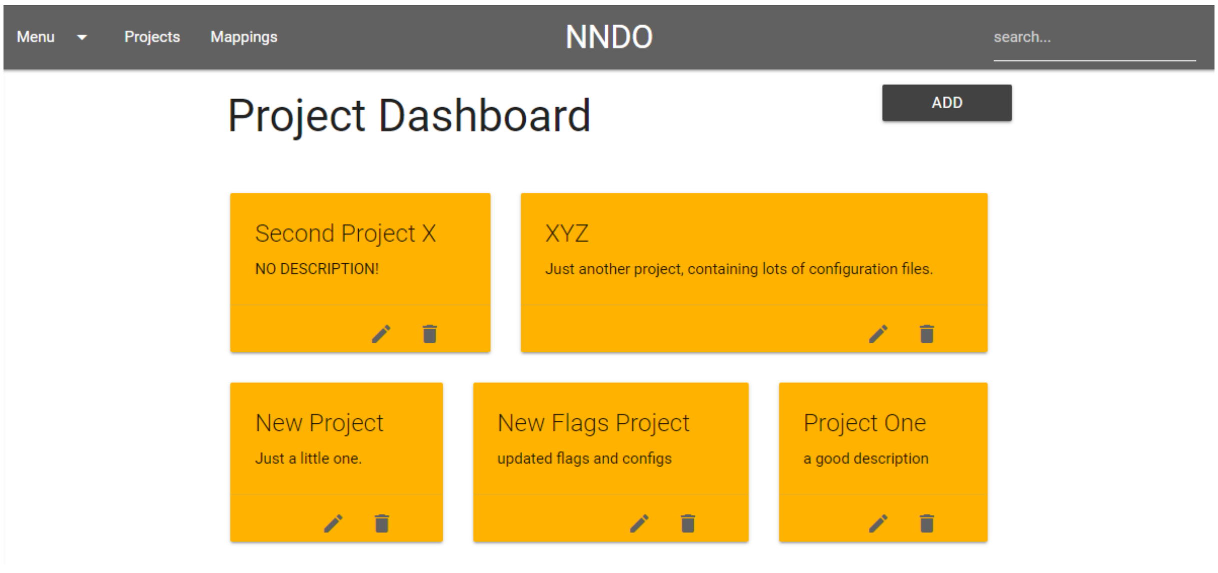
The account bound to this user is very important. Only the creator of a project, mapping or configuration file can edit or delete it. Any other users may see the work but cannot manipulate the data of others. In addition to those rights, any preferences stored are also bound to the user, which we will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

To log out, the user can to click on the menu button on the top left corner and press the log out 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.

3.4 Projects

The main page and the first page a user needs is the project dashboard (see Figure 6). 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.



Project Dashboard

Second Project X
NO DESCRIPTION!

XYZ
Just another project, containing lots of configuration files.

New Project
Just a little one.

New Flags Project
updated flags and configs

Project One
a good description

Figure 6 – The project dashboard with five project cards

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 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 7).

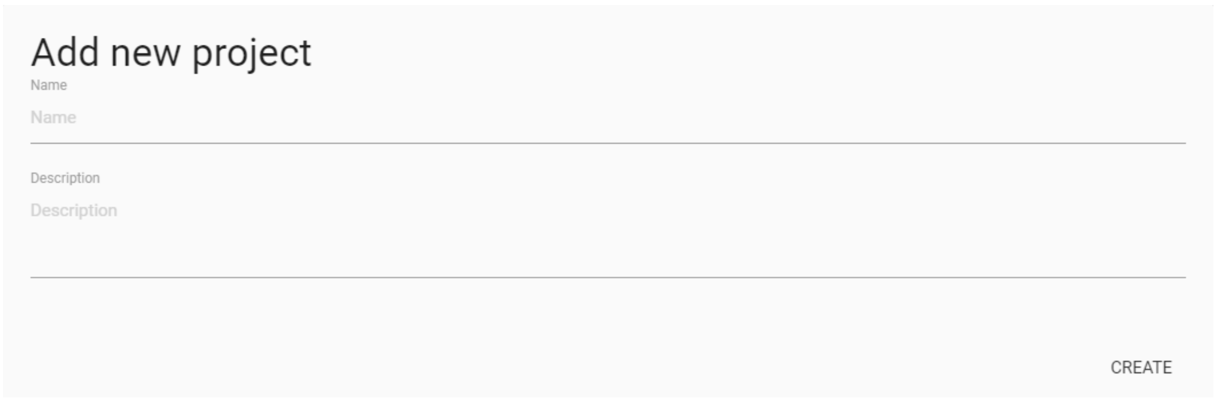


Figure 7 – The modal to add a new project

The user can now type in 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 creation is completed when 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 (see Figure 8).

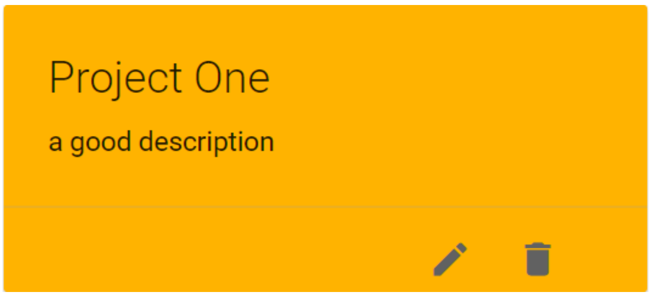


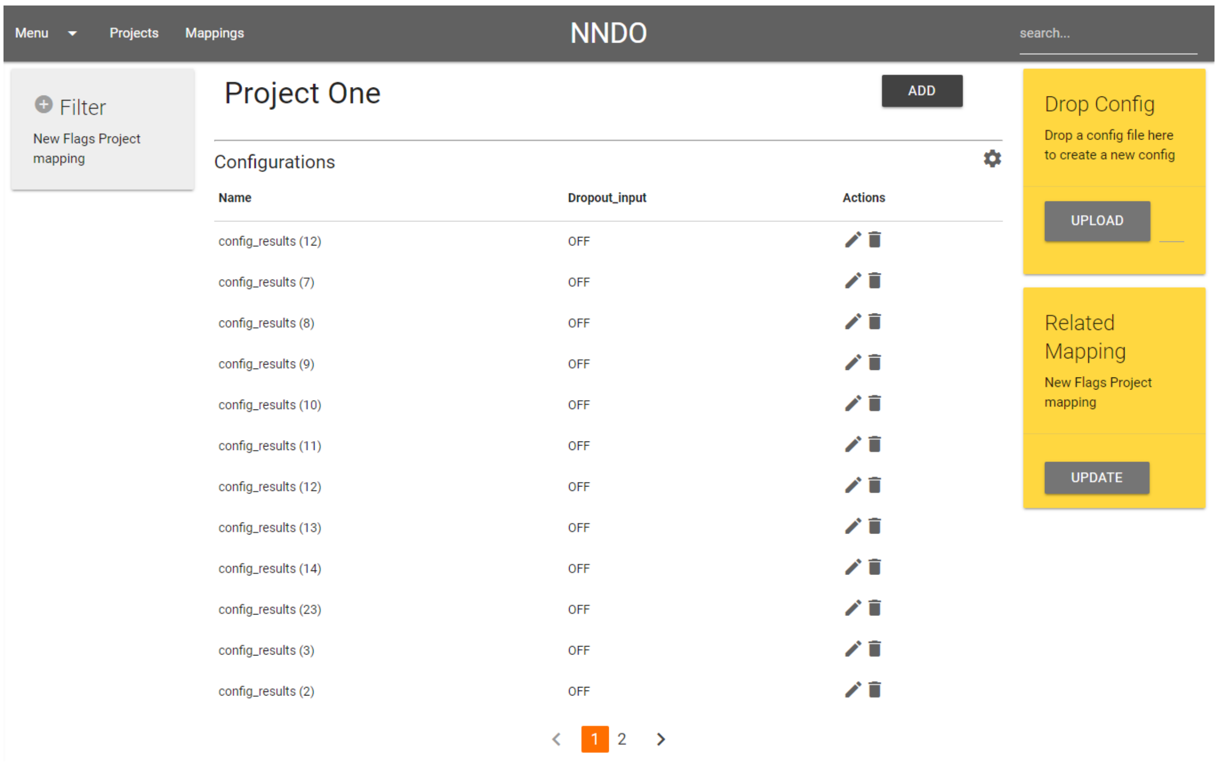
Figure 8 – A project card at the dashboard page

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 or description, the user has to click on the edit symbol. After checking if the user is the owner of the project and allowed to edit it, another modal will open, which looks similar to the creation modal, where the name and description 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 3). When he confirms the deletion, the project and all of the related configuration files will be deleted from the database.

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.

The next step is to upload configuration files, which can be done in any own project. To go to a projects page (see Figure 9), 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 or updated.



Filter

New Flags Project mapping

Project One

Configurations

Name	Dropout_input	Actions
config_results (12)	OFF	✏ 🗑
config_results (7)	OFF	✏ 🗑
config_results (8)	OFF	✏ 🗑
config_results (9)	OFF	✏ 🗑
config_results (10)	OFF	✏ 🗑
config_results (11)	OFF	✏ 🗑
config_results (12)	OFF	✏ 🗑
config_results (13)	OFF	✏ 🗑
config_results (14)	OFF	✏ 🗑
config_results (23)	OFF	✏ 🗑
config_results (3)	OFF	✏ 🗑
config_results (2)	OFF	✏ 🗑

Drop Config

Drop a config file here to create a new config

Related Mapping

New Flags Project mapping

Figure 9 – The project page with no active filter

3.5 Configurations

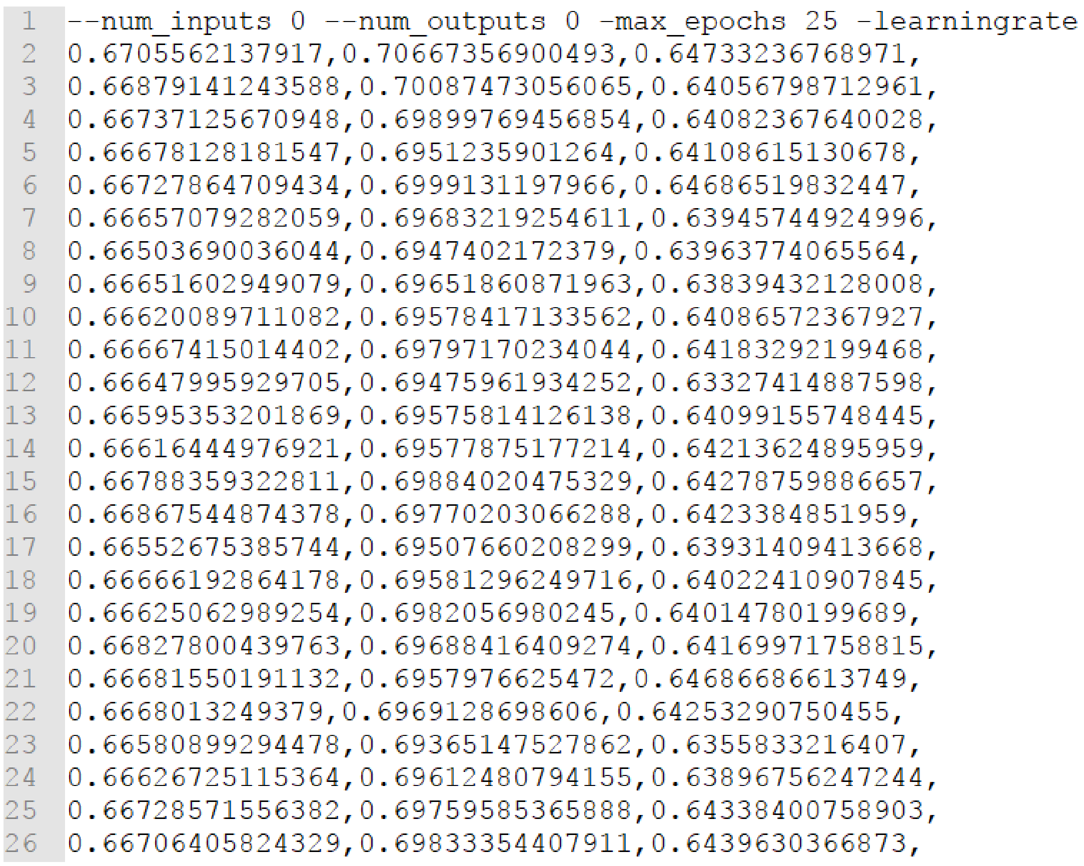
The configuration files can be uploaded by dragging them onto the *Drop Config Card* or clicking the upload button at a projects page (see Figure 9). If the application is able to find parameters with their values and results, the configuration file will be added as a *ConfigSet* to the database. By default, the uploaded file name will be chosen as the *ConfigSet* name and the file creation date is used as a description. Furthermore, the name and description of a *ConfigSet* can be edited by clicking on the edit button in the table.

3.5.1 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 with a white space.

All following lines represent the results. There can be as many training 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 like shown in Figure 10.



```
--num_inputs 0 --num_outputs 0 -max_epochs 25 -learningrate
0.6705562137917,0.70667356900493,0.64733236768971,
0.66879141243588,0.70087473056065,0.64056798712961,
0.66737125670948,0.69899769456854,0.64082367640028,
0.66678128181547,0.6951235901264,0.64108615130678,
0.66727864709434,0.6999131197966,0.64686519832447,
0.66657079282059,0.69683219254611,0.63945744924996,
0.66503690036044,0.6947402172379,0.63963774065564,
0.66651602949079,0.69651860871963,0.63839432128008,
0.66620089711082,0.69578417133562,0.64086572367927,
0.66667415014402,0.69797170234044,0.64183292199468,
0.66647995929705,0.69475961934252,0.63327414887598,
0.66595353201869,0.69575814126138,0.64099155748445,
0.66616444976921,0.69577875177214,0.64213624895959,
0.66788359322811,0.69884020475329,0.64278759886657,
0.66867544874378,0.69770203066288,0.6423384851959,
0.66552675385744,0.69507660208299,0.63931409413668,
0.66666192864178,0.69581296249716,0.64022410907845,
0.66625062989254,0.6982056980245,0.64014780199689,
0.66827800439763,0.69688416409274,0.64169971758815,
0.66681550191132,0.6957976625472,0.64686686613749,
0.6668013249379,0.6969128698606,0.64253290750455,
0.66580899294478,0.69365147527862,0.6355833216407,
0.66626725115364,0.69612480794155,0.63896756247244,
0.66728571556382,0.69759585365888,0.64338400758903,
0.66706405824329,0.69833354407911,0.6439630366873,
```

Figure 10 – A valid configuration file

3.5.2 ConfigSet Table

Every project's ConfigSet is displayed at the list or table in the center of the project page. This list can be filtered, which is explained in section 3.5.6, or personalized. 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 11).

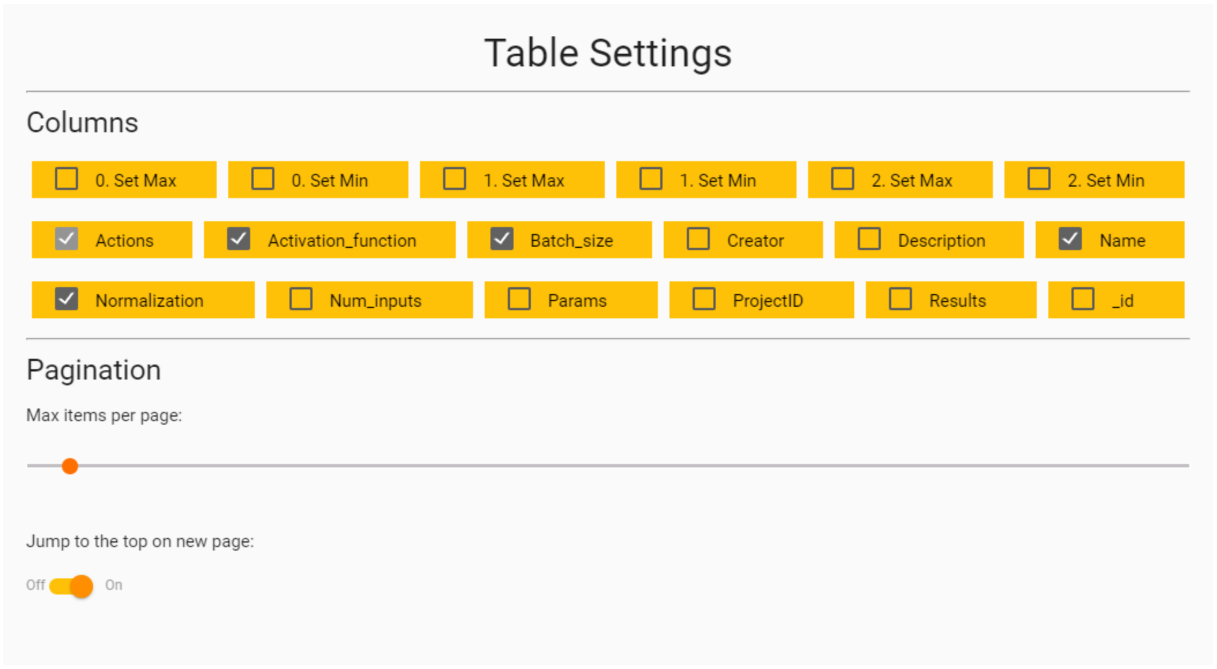


Table Settings

Columns

- [ ] 0. Set Max
- [ ] 0. Set Min
- [ ] 1. Set Max
- [ ] 1. Set Min
- [ ] 2. Set Max
- [ ] 2. Set Min
- [x] Actions
- [x] Activation_function
- [x] Batch_size
- [ ] Creator
- [ ] Description
- [x] Name
- [x] Normalization
- [ ] Num_inputs
- [ ] Params
- [ ] ProjectID
- [ ] Results
- [ ] _id

Pagination

Max items per page:

Jump to the top on new page:

Off [✓ On]

Figure 11 – ConfigSet table settings with column picker and pagination settings

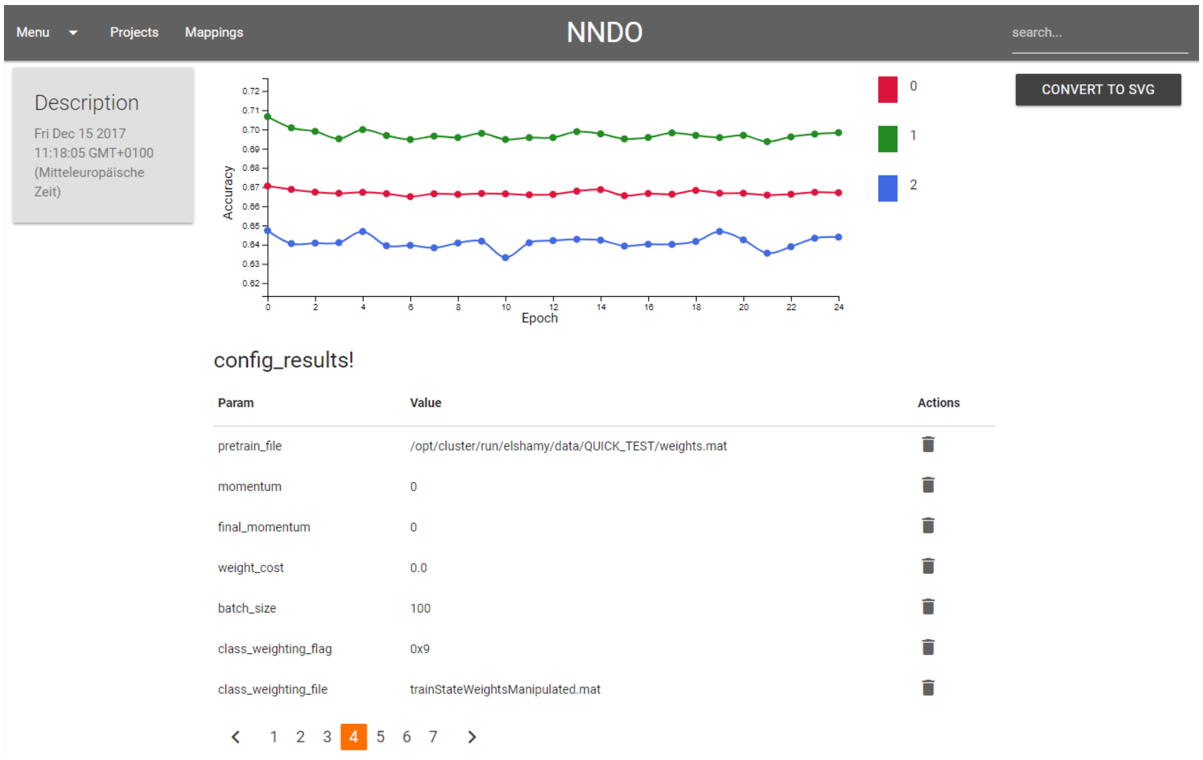
Every column, except for the 'Actions' column can be selected or deselected. The number of shown entries per page can be adjusted, as well as the option to jump to the top of the page, when opening a new one. The table settings are saved with the user's profile for every project.

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.

3.5.3 ConfigSet Page

At a *ConfigSet* page (see Figure 12) unnecessary parameters can be deleted and the results can be seen and extracted as a Scalable Vector Graphic (SVG).

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 accuracy and the x axis the epoch. Every training set is shown as a line. The chart diagram can be downloaded as a *SVG* file by first clicking on the convert button at the top right of the page and then clicking the appearing download button.



Description

Fri Dec 15 2017 11:18:05 GMT+0100 (Mitteleuropäische Zeit)

config_results!

Param	Value	Actions
pretrain_file	/opt/cluster/run/elshamy/data/QUICK_TEST/weights.mat	🗑
momentum	0	🗑
final_momentum	0	🗑
weight_cost	0.0	🗑
batch_size	100	🗑
class_weighting_flag	0x9	🗑
class_weighting_file	trainStateWeightsManipulated.mat	🗑

< 1 2 3 4 5 6 7 >

Figure 12 – A ConfigSet page

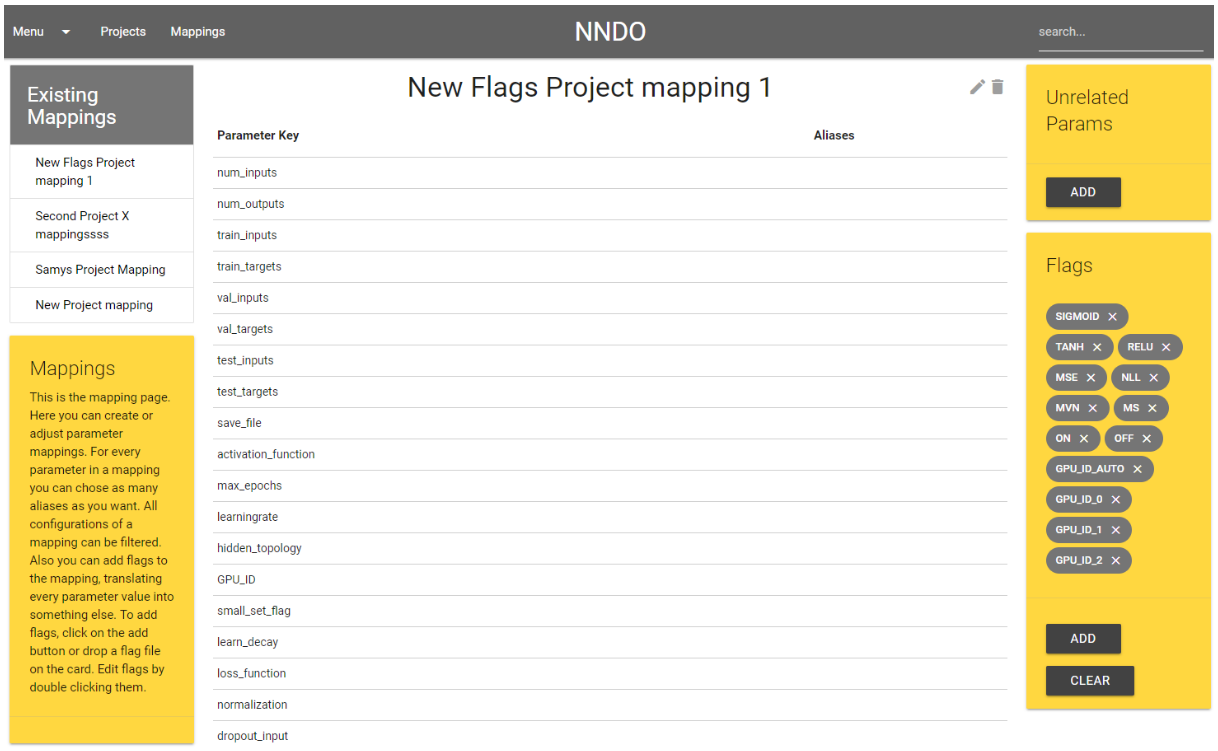
Every found parameter-value pair is shown at the bottom of the page. The table can be sorted by parameter or value by clicking on the column header. Each parameter-value pair that is not necessary can be deleted by clicking on the delete button if the current user is creator of the *ConfigSet*. After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

3.5.4 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. To accomplish that, a *Mapping* will assign aliases to parameters.

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 button of the mapping card at the project page. 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 key word or an alias or added as unrelated parameters, if they don't match any existing. When creating a new *Mapping*, the user will be informed via a toast message, how many unrelated parameters the *Mapping* has. There are two different ways of getting to the *Mapping's* page. The first one is by clicking on the 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 13) contains all parameters and aliases as well as flags, which are explained later (see 3.5.5).



Mappings

This is the mapping page. Here you can create or adjust parameter mappings. For every parameter in a mapping you can chose as many aliases as you want. All configurations of a mapping can be filtered. Also you can add flags to the mapping, translating every parameter value into something else. To add flags, click on the add button or drop a flag file on the card. Edit flags by double clicking them.

New Flags Project mapping 1

Parameter Key	Aliases
num_inputs	
num_outputs	
train_inputs	
train_targets	
val_inputs	
val_targets	
test_inputs	
test_targets	
save_file	
activation_function	
max_epochs	
learningrate	
hidden_topology	
GPU_ID	
small_set_flag	
learn_decay	
loss_function	
normalization	
dropout_input	

Figure 13 – A *Mapping* page without unrelated parameters

At the right of the page all 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 will aliases and the mapping's flags.

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. 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. 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.

3.5.5 Flags

Flags are used to give parameter values a different meaning. A flag holds a key and a meaning. Every found occurrence of the key will be translated to the meaning, 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 14). After hitting the create button, the new flag will be added to the mapping.

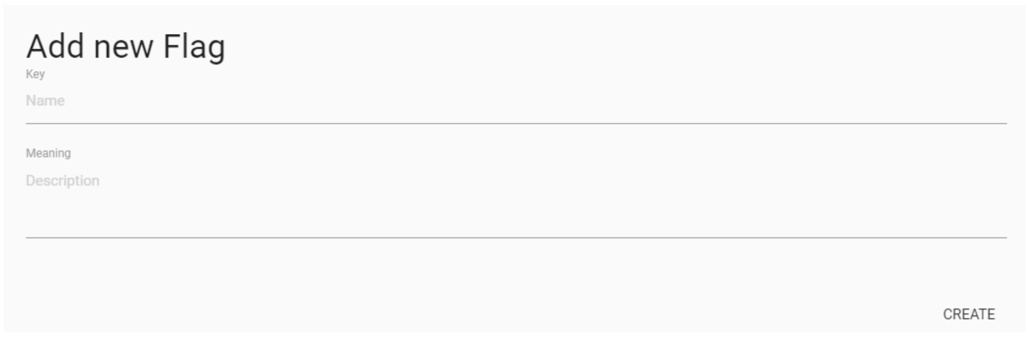


Figure 14 - Creating a new flag

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 15.

```
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 15 – A valid flag file

Flags can be deleted by clicking on the cross at the flag itself on the 'flag' card.

3.5.6 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 9). 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 16).

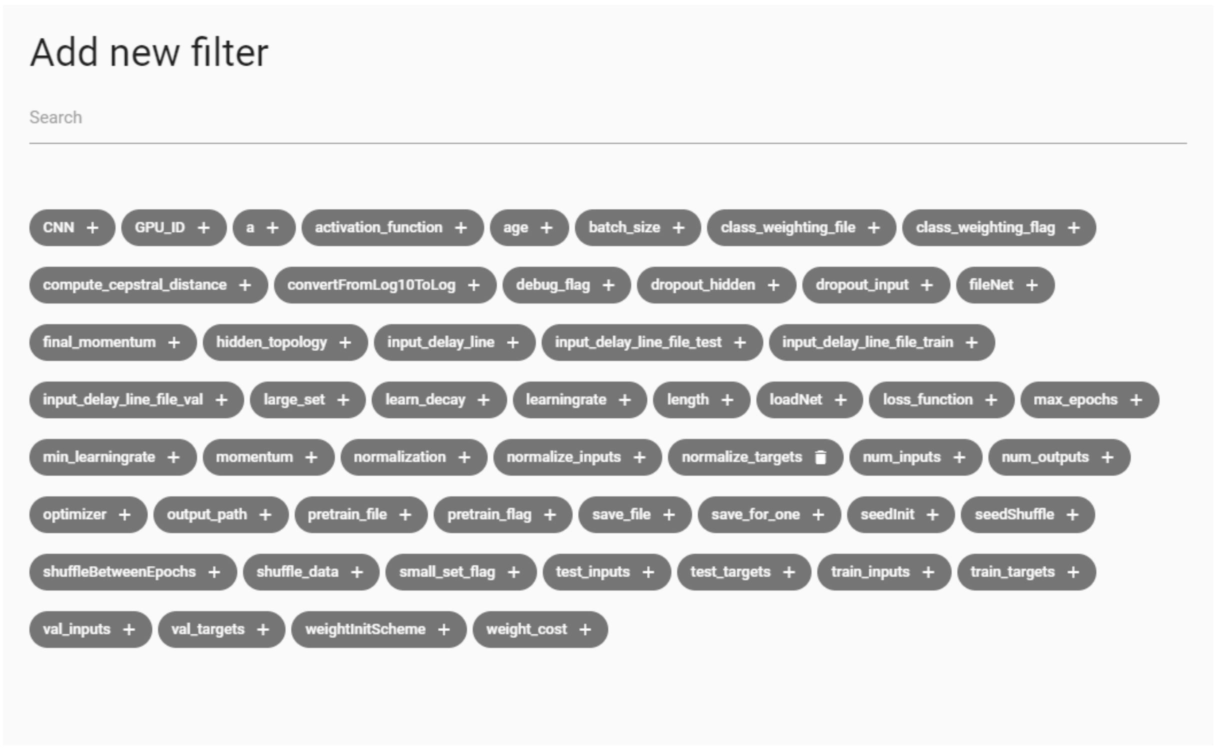


Figure 16 – The modal to add filters

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 16). 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 needed.

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.

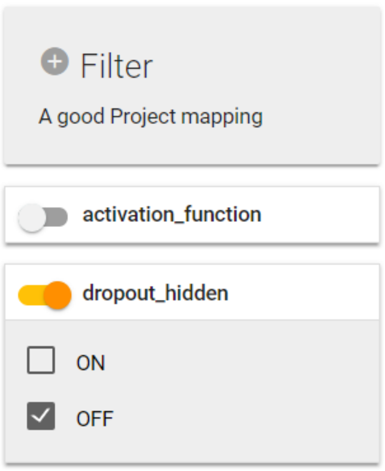


Figure 17 – The filter section with an active and an inactive filter

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 found 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 17 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 ConfigSets that do not have this option as a value for the referring parameter. This behavior 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.

4 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 [**A1**] and the Meteor [**M1**] website, going more into detail. Because Meteor-specific code is more straight forward, we will not be talking about it much.

4.1 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 [**N1**] and Meteor [**M1**], 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 [**WS1**], 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 [**AT1**] which can be very mighty with the right extensions.

Another helpful tool is Robo3T [**R1**] for viewing and editing the database, which isn't necessary for development.

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

4.2 Setting up a Development Environment

4.2.1 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 or NodeJS-Package-Manager (npm) packages have to be installed first. This can be done at 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 4.3.2). Now all missing meteor packages, which are listed in the *packages* 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.

4.2.2 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 ‘$USER_HOME$/AppData/Local/.meteor/meteor.bat’.

4.3 Overview

4.3.1 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. The third one, the *both* folder contains code, that the server as well as the client can see and work with. It holds things like models or 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 a SCSS file.

4.3.2 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 set 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.

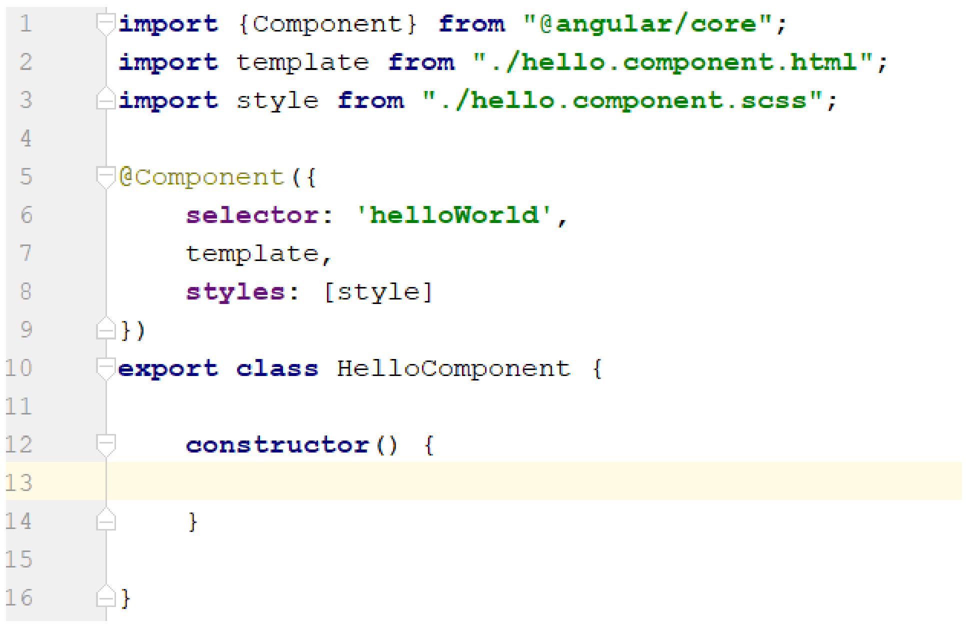
Those settings variables can be created at own discretion.

4.4 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 the 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 talking on how to add a new component and afterwards introducing some of Angular's features at the example of a component.

4.4.1 Creating and Declaring a Component

At first, a new folder with the name of the component should be created. The developer has to add an empty .ts, .html and .scss file to the folder. To implement a very basic component, only displaying a 'Hello World' text, the developer has to start by defining the component in the created .ts 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. This small component could look like shown in Figure 18.



```
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 18 – A basic typescript component class

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 detections, 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.

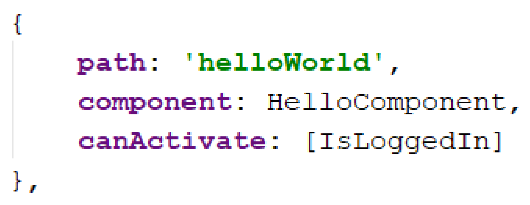
With these few steps the new component can be used in every other component, by using the selector (*<hello-world></hello-world>*) in a .html file. The displayed component would just be a plain text saying, ‘Hello World’.

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

4.4.2 Defining and Using Routes

If a component should become a new page with an own address, it needs a route. Routes are also defined at the *app.module.ts*. 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 22.



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

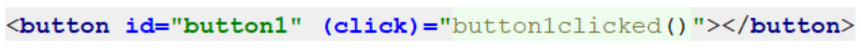
Figure 19 – An example route with guard

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 possible 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.

4.4.3 Using the Benefits of Angular

Angular provides easy-to-use functionality like mapping HTML events to the functions in a component. 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 20.

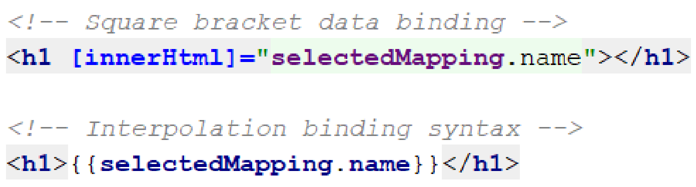


```
<button id="button1" (click)="button1clicked()"></button>
```

Figure 20 – Linked button click to a component function

The round brackets around the attribute is Angular's way to declare data-binding from the template to the component. So, when the HTML event 'click' is triggered, it calls the method given in the template.

Square brackets 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 without any delay. For this example the same effect can be achieved through Angular's *interpolation binding* syntax, which are two curly brackets around an expression, like a variable name. Figure 21 shows the two different approaches, which lead to the same result.



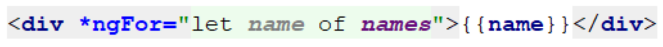
```
<!-- Square bracket data binding -->
<h1 [innerHtml]="selectedMapping.name"></h1>

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

Figure 21 - A component variable expressed in two ways

4.4.4 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 of them 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 22. Whenever the 'names' variable changes, the displayed list will also change immediately.



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

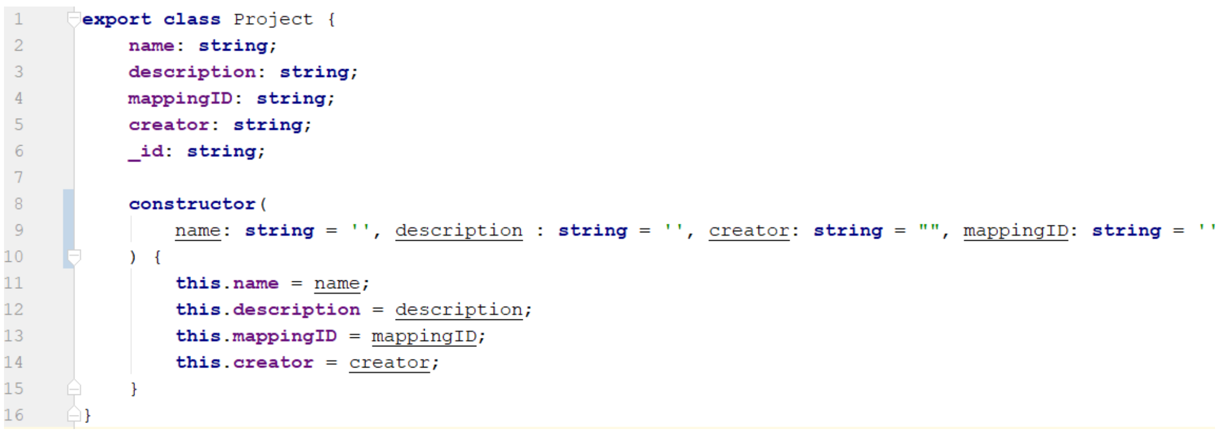
Figure 22 – An example of the *ngFor directive

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.

4.5 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 at 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 23.

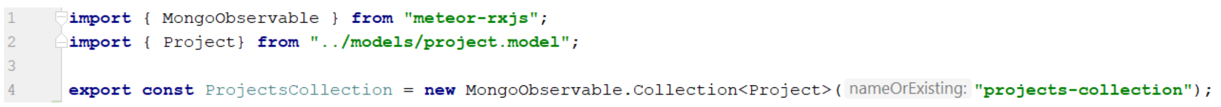


```
export class Project {
    name: string;
    description: string;
    mappingID: string;
    creator: string;
    _id: string;

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

Figure 23 – A typical collection’s object class definition

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 24 shows the complete *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 24 – Definition of a collection

The project class is imported, to declare the type for the stored objects, as well as a *Mon-goObservable* is imported. 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.

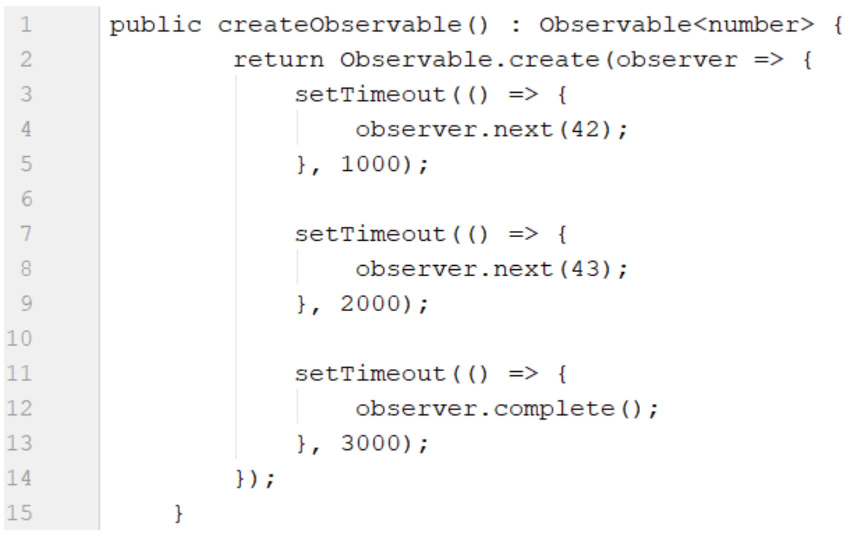
4.6 Data Services and Observables

All data actions for our application are handled in the data service, which is responsible for the corresponding collection. When getting data from the databank, it takes a bit of time to fetch all results and prepare them for the client, so it can use 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 Anuglar's dependency injection, which we mentioned earlier.

4.6.1 Observables

An Observable is used for handling an asynchronous data steam, 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 [**RX1**].

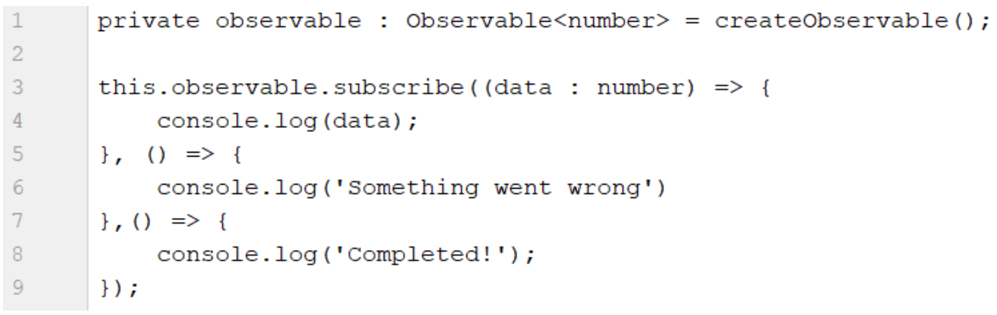
MongoDB and the server can't handle every request the client is sending instantly. Instead of active waiting, notifying the client when the data is ready is better. In our application this is done by RXJS' observables.



```
public createObservable() : Observable<number> {
        return Observable.create(observer => {
            setTimeout(() => {
                observer.next(42);
            }, 1000);

            setTimeout(() => {
                observer.next(43);
            }, 2000);

            setTimeout(() => {
                observer.complete();
            }, 3000);
        });
    }
```

Figure 25 – A simple observable returning two numbers

Figure 25 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 25. 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 compete 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 26.



```
private observable : Observable<number> = createObservable();

this.observable.subscribe((data : number) => {
    console.log(data);
}, () => {
    console.log('Something went wrong')
},() => {
    console.log('Completed!');
});
```

Figure 26 – A basic observer with ‘next’, ‘error’, and ‘completed’ function

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.

4.6.2 Data Services

4.7 Extending Functionality

-Npm packages

4.8 Documentation

4.9 Deployment

A Appendix

B Bibliography

[M1] © 2017 Meteor Development Group Inc. (2017) Meteor.com. [Online]. https://www.meteor.com/

[A1] Google Inc. (2017) Angular. [Online]. https://angular.io/

[Mo1] MongoDB, Inc. (2017) MongoDB. [Online]. https://www.mongodb.com/

[N1] Node.js Foundation. (2017) NodeJS. [Online]. https://nodejs.org/en/

[G1] Google Inc. (2017) Google. [Online]. www.google.de

[ECM17] ECMA international. (2017) json.org. [Online]. https://www.json.org/

[E1] ECMA international, ISO/IEC 16262:2011, 2011-06, ISO/IEC 16262:2011 defines the ECMAScript scripting language.

[T1] Microsoft Corporation. (2017) Typescript. [Online]. https://www.typescriptlang.org/

[SA1] Natalie Weizenbaum, Chris Eppstein Hampton Catlin. (2017) SASS. [Online]. http://sass-lang.com/

[RX1] (2017) ReactiveX. [Online]. http://reactivex.io/

[WS1] JetBrains. (2017) Webstorm. [Online]. https://www.jetbrains.com/webstorm/

[AT1] Open Source. (2017) Atom. [Online]. https://atom.io/

[R1] 3T Software Labs GmbH. (2017) Robo3T. [Online]. https://robomongo.org/

C List of Figures

D List of Tables

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.