Implementation of an improved database concept for the storage and access of analytical data and development of a user interface for data entry and retrieval.

Bachelor Thesis

Bachelor of Engineering

of the Course of Studies Electrical Engineering, discipline Automation

at the Cooperative State University Baden-Württemberg Stuttgart

by

Lukas Veit

07/09/2020

Time of Project	15/06/2020 - 07/09/2020
Student ID, Course	5999138, TEL17GR3
Company	Robert Bosch GmbH, Renningen
Supervisor in the Company	Dr. Elisabeth Lotter
Reviewer	Prof. Dr. Janko Dietzsch

Cooperative State University Baden Württemberg
Major of Study Engineering,
branch of study Electrical Engineering

Practical Training Project Report of the 6 th term of study

Name: Lukas Veit

Starting Date of Study: 2017

Training Department: CR/APS1 Corporate Research, Analytics and Prototype Sample Shop

Location: Renningen

Duration - Start: 15/06/2020 End: 07/09/2020

Job Description: Implementation of an improved database concept for the storage and access of analytical data and development of a user interface for data entry and retrieval.

Supervisor: Dr. Elisabeth Lotter

Supervisors Certification:

I hereby certify, that the following report has been reviewed and the contents are both factual and technically accurate.

Location Date Department, Signature

Certification by student:

According to §5(3) of „Studien- und Prüfungsordnung DHBW Technik“ (September 29th, 2015): I hereby certify, that I am the original author of the following report. All the information provided are only from the sources listed under the bibliography section.

Location Date Department, Signature

Author's declaration

Hereby I solemnly declare:

1. that this Bachelor Thesis, titled *Implementation of an improved database concept for the storage and access of analytical data and development of a user interface for data entry and retrieval.* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Bachelor Thesis has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Bachelor Thesis in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Stuttgart, 07/09/2020

Lukas Veit

Confidentiality Statement

The Bachelor Thesis on hand

Implementation of an improved database concept for the storage and access of analytical data and development of a user interface for data entry and retrieval.

contains internal respective confidential data of Robert Bosch GmbH. It is intended solely for inspection by the assigned examiner, the head of the Electrical Engineering, discipline Automation department and, if necessary, the Audit Committee at the Cooperative State University Baden-Württemberg Stuttgart. It is strictly forbidden:

- to distribute the content of this paper (including data, figures, tables, charts etc.) as a whole or in extracts,
- to make copies or transcripts of this paper or of parts of it,
- to display this paper or make it available in digital, electronic or virtual form.

Exceptional cases may be considered through permission granted in written form by the author and Robert Bosch GmbH.

Stuttgart, 07/09/2020

Lukas Veit

Abstract

The project DeFinMa aims to provide methods to classify plastic materials and material properties with machine learning models using infrared spectroscopy. The task of this thesis was to improve the project's data management. This should increase uniformity for easier development as well as make uploading data for departments, which contribute data, more convenient.

The existing data was analyzed and a fitting database model was implemented on a MongoDB instance. All existing data was imported into the database. Access to this data is provided via an API hosted on a Node.js server. This server manages access rights and validates incoming data. Database collections are joined and filter and sort possibilities are provided for convenient data access.

This work also included a web app for a graphical interface. Next to user management for access control it includes a data overview with filter possibilities, allowing developers to extract exactly the data needed and download them using a generated link. Sample upload and edit was also implemented as well as a dialog for maintainers allowing to manage the templates used in the data structure. A page to predict properties by developed machine learning models was also implemented.

Database, API and web app were all completely finished and are now the main method to manage data in the DeFinMa project.

Abstract

Das Projekt DeFinMa arbeitet daran, Methoden zur Klassifizierung von Kunststoffen und Materialeigenschaften anhand von Infrarotspektroskopie mittels Machine Learning zu entwickeln. Die Aufgabe dieser Arbeit bestand darin, das Datenmanagement des Projekts zu verbessern. Dies sollte die Einheitlichkeit der Daten erhöhen, um die Entwicklung zu erleichtern und das Hochladen von Daten für Abteilungen, die Daten beisteuern, einfacher zu machen.

Die vorhandenen Daten wurden analysiert und ein angepasstes Datenbankmodell wurde auf einer MongoDB-Instanz implementiert. Alle vorhandenen Daten wurden in die Datenbank importiert. Der Zugriff auf diese Daten erfolgte über eine API, die auf einem Node.js Server gehostet wird. Dieser Server verwaltet die Zugriffsrechte und validiert eingehende Daten. Die Datenbank Collections werden kombiniert und Filter- und Sortiermöglichkeiten für einen bequemen Datenzugriff bereitgestellt.

Diese Arbeit beinhaltet auch eine Webanwendung als grafische Oberfläche. Neben der Benutzerverwaltung zur Zugriffskontrolle enthält sie eine Datenübersicht mit Filtermöglichkeiten, die es Entwicklern erlaubt, die benötigten Daten genau zu extrahieren und über einen generierten Link herunterzuladen. Der Upload und die Bearbeitung von Proben wurden ebenso implementiert, wie ein Dialog für Datenbankverwalter, der es erlaubt, die in der Datenstruktur verwendeten Vorlagen zu verwalten. Eine Seite zur Vorhersage von Eigenschaften durch entwickelte Machine Learning Modelle wurde ebenfalls implementiert.

Datenbank, API und Webanwendung wurden alle vollständig fertiggestellt und sind nun die Hauptmethode zur Verwaltung der Daten im Projekt DeFinMa.

Contents

Acronyms

API	**A**pplication **P**rogramming **I**nterface
BGN	**B**osch **G**lobal **N**et
BIC	**B**osch **I**oT **C**loud
BSON	**B**inary **JSON**
CRUD	**C**reate **R**ead **U**pdate **D**elete
CSP	**C**ontent **S**ecurity **P**olicy
CSS	**C**ascading **S**tyle **S**heets
CSV	**C**omma-**S**eparated **V**alues
DeFinMa	**De**coding the **Fin**gerprint of **Ma**terials by AI
DNS	**D**omain **N**ame **S**ystem
DOM	**D**ocument **O**bject **M**odel
DPT	**D**ata **P**oint **T**able
DTD	**D**ocument **T**ype **D**efinition
HTML	**H**yper**t**ext **M**arkup **L**anguage
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
HTTPS	**H**yper**t**ext **T**ransfer **P**rotocol **S**ecure
IoT	**I**nternet **o**f **T**hings
IP	**I**nternet **P**rotocol
JS	**J**ava**S**cript
JSON	**J**ava**S**cript **O**bject **N**otation

MIME	**M**ultipurpose **I**nternet **M**ail **E**xtensions
OAS	**O**pen**A**PI **S**pecification
PDF	**P**ortable **D**ocument **F**ormat
PHP	**P**HP: **H**ypertext **P**reprocessor
Regex	**Reg**ular **ex**pression
ReST	**Re**presentational **S**tate **T**ransfer
SQL	**S**tructured **Q**uery **L**anguage
SVG	**S**calable **V**ector **G**raphics
TS	**T**ype**S**cript
UI	**U**ser **I**nterface
URI	**U**niform **R**esource **I**dentifier
URL	**U**niform **R**esource **L**ocator
VN	**V**iscosity **N**umber
XML	E**x**tensible **M**arkup **L**anguage
XSS	**Cross S**ite **S**cripting
YAML	**Y**AML **A**in't **M**arkup **L**anguage

List of Figures

List of Tables

Listings

1 Introduction

With an increasing demand for plastic parts and the resulting expansion of the plastics industry, resources are becoming scarce and unauthorized material changes happen more often, possibly leading to fatal incidents due to changed material properties. The project **De**coding the **Fin**gerprint of **Ma**terials by AI (DeFinMa) aims to provide methods to classify and predict plastic materials and material properties and therefore assess material quality in a fast and cost-effective way using infrared spectroscopy and existing engineering data. Based on collected data machine learning models are developed to predict the required properties. [8]

1.1 Motivation

As with many machine learning tasks, a good prediction model can only be developed with a large amount of consistent data for model training. An additional challenge for the DeFinMa project is the diversity of plastic materials. Materials with a different chemical structure result in a different spectrum. To predict a property of a material, a dedicated prediction model has to be developed for this material which again requires its own data for training.

To speed up the data acquisition and get more usable data, cooperations with departments from Bosch plants were made, which already measured spectra as well as target properties for other purposes, and contributed their data to the project.

The whole data management was done via a file share until now. All metadata of the samples is stored in a spreadsheet and each sample is given a number. The spectrum measurements are each stored in a separate file, whose filename contains the sample number of the sample the spectrum is belonging to. This process has a few disadvantages. As every contributor needs access to the file share, the data is stored very insecurely. Every user, who was granted access, can copy, rename and delete all data, either by accident or by purpose. All input is also very susceptible to user error. A spreadsheet has no automatic

data checks implemented, which allows for contradictory data input like a product name which does not correspond to the entered material. To make use of the collected data for machine learning, uniformity is also a key point. Spelling mistakes and different spellings of, for example material name or supplier, make automated data extraction very difficult. This also applies to the spectrum names. While there is given a dedicated naming format in the beginning, users forget this format or include further information in the file name, which results in errors when matching the file name with a sample number and requires a lot of manual correction.

As previously stated, the collected data and the size of it is of enormous value for a machine learning application like the DeFinMa project. If the data gets messed up, data collection has to start from zero, resulting in fatal consequences for the project.

1.2 Task

This work aims to improve the current data management situation. All sample data should be stored in a database, which, as the name suggests, provides a lot of benefits for managing data. Therefore a database schema, fitting the available data, should be developed and all currently available data should be imported into the database.

An **A**pplication **P**rogramming **I**nterface (API) for data access should be implemented. The task of this API is to provide access to the database in an easy and secure way, giving each user only the respective permissions necessary. Incoming data should also be validated to ensure consistent data sets, which can easily be used in machine learning tasks.

To allow an easier and more user-friendly way to access the data, a web app should be developed. It should provide an easy and versatile way to view existing data, suited for new customers as well as developers to select the right data needed for their machine learning development. The upload for other departments should also be simplified, while user errors like spelling mistakes should be mitigated. Finally the web app should also provide a quick way to distribute developed and trained machine learning models among customers to try the prediction capabilities.

While the whole architecture should only be available inside the **B**osch **G**lobal **N**et (BGN) for now, with the prospect of making the service accessible to third parties on the internet, application security should be considered throughout all parts. Whereas the BGN gives some protection against attackers, the created application should be ready to be hosted publicly when the need arises, without major security modifications.

1.3 Methodology

As already shown in the preceding work [10], for further database development the method described by Atzeni et al. [1] was used.

The API was designed using common best practices as for example described in [15]. This includes aspects like logic **U**niform **R**esource **L**ocator (URL) naming: While `/users` lists all users, `/users/johndoe` lists the properties of the user *johndoe*, making the structure easy to understand. Another aspect is the correct use of **H**yper**t**ext **T**ransfer **P**rotocol (HTTP) features like using HTTP methods to describe the kind of action which is performed or using the correct HTTP response code to tell the API user the outcome of the request.

The Angular front end was mainly developed using the official Angular guides [25], which exhaustively describe all Angular features and best practices. The Angular concepts, which are described in more detail in section 2.3, were applied to create a modular and efficient web app.

2 Technical basics

2.1 Used programming and markup languages

2.1.1 HTML

Hyper**t**ext **M**arkup **L**anguage (HTML) is a markup language to describe web pages. In an HTML document the web page is built up from individual elements to describe the structure of the page. Each element is declared by using the corresponding HTML tag like `<html>`. Most elements consist of an opening `<html>` tag and a closing `</html>` tag, between which the content of the element or further nested elements are declared. As seen in line 9 of Listing 2.1 not all elements like the `<input>` need a closing tag as they cannot nest any additional content inside. HTML tags can further possess attributes specifying the properties of the element like the `type="text"` attribute declaring the type of input which this input element expects. The attributes `class=""` and `id=""` are used for referencing elements. Thereby a `class` can be given to multiple elements and an element can have multiple classes whereas one `id` may only be used for one element.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Website Title</title>
5      <link rel="stylesheet" type="text/css" href="styles.css">
6    </head>
7    <body>
8      <h1 class="heading accent">Website Heading</h1>
9      <input type="text" value="Input value" id="input0">
10   </body>
11 </html>
```

Listing 2.1: HTML base structure of a web page

Listing 2.1 shows the base structure of every HTML web page. The document is started with the **D**ocument **T**ype **D**efinition (DTD) to declare the HTML version of the document which in this case is HTML5. The DTD is followed by the actual HTML document between the `<html></html>` tags. It is structured into `<head>` and `<body>`. The `<head>` is not visibly rendered but rather used to declare properties of the page like the title or further documents to load, like style sheets or scripts. The `<body>` describes the part of the web page visible for the user. The browser interprets the declared elements and displays them accordingly. In the given example the content inside the `<h1>` is rendered as a heading and the `<input>` is rendered as a user text input field which is initialized with *Input value* as specified in the `value` attribute. [22]

2.1.2 CSS

The look of HTML elements is described by **C**ascading **S**tyle **S**heets (CSS). Therefore a group of elements is selected to apply a specific set of styles to this group. Selection can be done using the HTML tag name like in line 1 of Listing 2.2, using the class by adding a `.` before the class name in line 5 or using the id by adding a leading `#` as seen in line 9. If multiple selectors are given, an element has to match all.

For the selected group the style rules are declared. These can be simple rules like background color, font weight or the element width shown in Listing 2.2, but also more complex rules describing the layout or animations. The style sheet is added to the page by specifying the source in the HTML document, shown in line 5 of Listing 2.1. The browser then automatically downloads the style sheet and applies the rules as given. [18]

```
1  body {
2  background: aqua;
3  }
4
5  .heading .accent {
6  font-weight: bold;
7  }
8
9  #input1 {
10 width: 200px;
11 }
```

Listing 2.2: Example of a CSS document

The finished page consisting of HTML and CSS looks as follows:

Figure 2.1: Displayed page from the code examples above

2.1.3 JS

Java**S**cript (JS) is a programming language which can be executed in the browser. It is interpreted and object oriented, but does not have classes. In the browser JS enables dynamic web pages, allowing the programmer to apply changes to the currently displayed HTML. Furthermore background activity like sending and receiving HTTP requests is possible.

2.1.4 TS

Type**S**cript (TS) is based on JS and extends it by variable types and classes. JS code is fully compatible with TS. TS code cannot be directly executed like JS but rather has to be compiled into JS before execution. [20]

2.1.5 JSON

Java**S**cript **O**bject **N**otation (JSON) is a data interchange format. It allows for encoding arrays of values surrounded by `[]`, and objects surrounded by `{}`, consisting of values with a key in string format. The allowed values are strings in `""`, numbers which can be in exponent notation, `true`, `false` and `null`. [23]

```
{
  "name": "value",
  "list": [true, 1e3, {"name": null}]
}
```

Listing 2.3: Example of a JSON document

2.2 Node.js

Node.js is a runtime environment for JS. While JS was primarily developed for usage in the browser, it is common today to use JS on the server side, too. Node.js is event based and asynchronous. Functions are executed as a reaction to a certain event, which is often being a client request. Being asynchronous is important, as the server thread is not completely blocked when the server is waiting for a response. Instead it is able to accept further client requests in the meantime and processes the response once it has arrived.

The concept of Node.js is contrary to a standard synchronous **P**HP: **H**ypertext **P**reprocessor (PHP) server, where an application like Apache handles all incoming requests, executes the PHP script for the requested path and returns the response to the client. Node.js handles both of these layers: The Node.js server is running constantly, receiving and handling all requests directly and responding to the client. This allows for a much more dynamic route handling, making it easy to include parameters at any point of the route. It also enhances code sharing between different routes and central error handling for database errors or *Not found* pages.

Handling of client requests can be done on a high level by using modules like express, which provide the base for setting up a web server, allowing you to easily specify event handlers for a specific route. [11]

2.3 Angular

Angular is a frontend web application framework. It provides a base and tools for easier development of web apps and is developed mainly in TS.

Angular apps are developed on a Node.js server which allows hosting the application for development and testing it in the browser. This server is also used for Angular development tools for code generation and testing. For deployment, stand-alone HTML, CSS and JS documents are compiled from the code source, which allows the app to be hosted on any server which can serve static files.

Angular apps are single page applications. The whole app, including all subpages consists of a single web page. Navigation feels usual for the user and the URL changes accordingly, but the according subpage is not downloaded for every change to another page. Instead all information to be displayed on the subpages is downloaded on the first app access. The requested subpage is then constructed in the browser using JS and eventually additionally

needed data is loaded in the background. Using this technique, the initial page load might take a bit longer than for a normal web page as not only visible information is loaded but also information for subpages the user might not want to visit, but navigation between subpages is faster as all needed information is already downloaded. The use of service workers allows caching the static files, which increases load times on subsequent visits. [13]

Angular apps consist of a number of Angular-specific elements. All these elements are TS classes which are declared as a specific element by using Decorators. Such a Decorator can be seen in lines 3-7 of Listing 2.4: It prefixes the class and always begins with an `@`, followed by the element name and the arguments for the element. During compilation the decorator is converted to a function call defined in the Angular framework to which the arguments as well as the constructor of the class are passed to. [20]

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4  selector: 'app-sample',
5  templateUrl: './sample.component.html',
6  styleUrls: ['./sample.component.scss']
7  })
8
9  export class SampleComponent implements OnInit {
10 ngOnInit() { }
11
12 values = [1, 2, 3];
13
14 logValue(v) {
15 console.log(v);
16 }
17 }
```

Listing 2.4: TS code example of a `sample` Component

2.3.1 Component

Components are the elements which are directly responsible for displaying content. A Component consists of an HTML Template defining the elements to display, a TS class holding all logic, a style sheet for the Component's style rules as well as a test file for unit testing.

An example for the base structure of a Component class can be seen in Listing 2.4. In line 1 the necessary Angular modules have to be loaded, followed by the Decorator in lines 3-7, declaring this class to be an Angular Component. Here the HTML selector is specified which is used to include this component into another Component's HTML template, further the paths to the corresponding template and style documents are set. The actual class starts in line 9 and implements `OnInit`, which is an Angular life cycle hook. These allow performing certain actions at specific points in the life cycle of the Component. `ngOnInit()`, for example, is executed when the Component is initialized, whereas `ngOnDestroy()` is executed when the Component is destroyed because it is no longer visible. From line 12 on the actual code of the Component starts with declarations of attributes and methods, which are needed to be displayed in the template or to be executed in response to a user action like clicking a button.

```
1  <h1>Sample Component</h1>
2  <p>First Value: {{values[0]}}</p>
3  <div> All Values:
4    <ul>
5      <li *ngFor="let value of values">
6        <input (input)="logValue(value)" [(ngModel)]="value">
7      </li>
8    </ul>
9  </div>
10 <app-subsample [test]="value[2]"></app-subsample>
```

Listing 2.5: Code example of a Component template

Listing 2.5 shows the corresponding template to Listing 2.4. Templates use HTML elements but extend the normal HTML syntax to connect the data of the class with the HTML elements. Line 1 and 2 for example show normal HTML elements, but the content of the `<p>` tag is not normal text, but text mixed with the content of the variable `values[0]` which is embedded by using `{{ }}`, known as an Angular interpolation. Property binding allows setting variable values for normal HTML properties by wrapping the attribute inside `[ ]`. Angular allows for two-way binding: data cannot only get from the class into the template but also back from the template into the class by using Event Binding, which uses a `( )` syntax like in line 6, when the input event is used to execute a method. Binding of a variable can even be done bi-directional using `[(ngModel)]`: The displayed value is updated when the variable value is changed, but when the user changes the value of the input field, the variable value is changed accordingly.

Another useful function of Angular templates are structural directives like `*ngFor` in line 5. These can alter the normal display of an element. In the example, the `*ngFor` directive iterates over the `values` array and displays the `<li>` element as well as everything nested inside multiple times, providing the value of the current array item for the elements inside. There also exist other structural directives like `*ngIf`, which displays the element only if the given condition is true.

To specify where the complete Component should be displayed, the Component is used in another template by using the selector like the `<app-subsample>` in line 10. Starting from the root Component, Components can be nested inside each other, together forming a single view. [13]

2.3.2 Service

A Service only consists of a testing file and a TS class, shown in Listing 2.6. Services are used to implement logic which is not directly needed to display content or which is shared between multiple Components. The structure is similar to the one of a component, first importing the necessary modules, followed by the `@Injectable` Decorator. The Decorator argument `providedIn` describes the scope in which this service can be used, in this case in the root component and every Component nested at any level inside it. In this case there exists a single class instance for all Components using this Service. However if the scope of the Service were limited to a certain Component, a new Service instance would be created for every Component instance. Starting in line 8, the actual methods needed for the Service can be written. [13]

```
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4  providedIn: 'root'
5  })
6
7  export class SampleService {
8
9  constructor() { }
10 }
```

Listing 2.6: Code example of a Service

2.3.3 Module

For bigger Angular apps it is preferable to structure the code into multiple Modules for reusability. Each Module consists of Components and Services as well as a Module definition. An Angular project needs at least a root module, whose definition can be seen in Listing 2.7. Next to the Angular module all modules and components are imported, which are necessary for the app. The definition of the Module takes place entirely in the Decorator, the class itself is always left empty and only included for syntactic reasons. In the Decorator, `declarations` lists all Components declared in this Module. `imports` specifies all Modules to import for usage inside this Module, whereas `exports` specifies the classes to export from this Module. `providers` are needed to declare services which are declared in this Module but should be available throughout the whole app. The `bootstrap` property may only be declared inside the root Module, specifying the root Component, inside which all other Modules and Components are nested. [13]

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { FormsModule } from "@angular/forms";
4  import { AppComponent } from './app.component';
5  import { SampleComponent } from './sample/sample.component';
6
7  @NgModule({
8  declarations: [
9  AppComponent,
10 SampleComponent
11 ],
12 imports: [
13 BrowserModule,
14 FormsModule
15 ],
16 exports: [],
17 providers: [],
18 bootstrap: [AppComponent]
19 })
20 export class AppModule { }
```

Listing 2.7: Code example of the `rootModule` definition

2.4 OAS

The **O**pen**A**PI **S**pecification (OAS) describes a uniform format to describe **Re**presentational **S**tate **T**ransfer (ReST) APIs. An API in general is an interface between two programmes. In this context an API is generally referred to an interface, which is defined and provided by a server, allowing clients to exchange information using specific methods. ReST defines the type of communication: Here every request is self-contained and the API itself is stateless. Therefore a request is independent from preceding requests and has to include all information necessary for the server to fulfil the request.

An OAS consists of a single JSON or **Y**AML **A**in't **M**arkup **L**anguage (YAML) document which can be separated into multiple sub-documents using references. The following examples use YAML instead of JSON as it provides better clarity using indentations instead of braces and `-` to declare arrays. There are libraries like `swagger-ui` to generate a web page from an OAS, listing all methods and interfaces and even providing the possibility to test the API routes like shown below. [28]

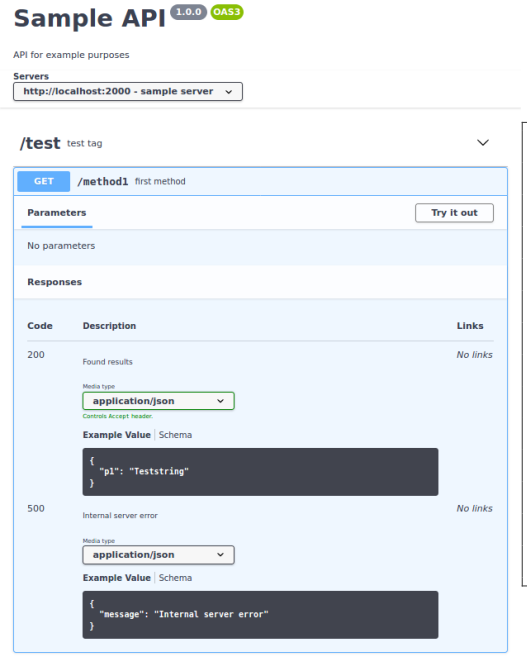

Figure 2.2: Generated OAS web page

```yaml
openapi: 3.0.2

info:
  version: 1.0.0
  title: Sample API
  description: API description

servers:
  - url: http://localhost:2000
    description: sample server

tags:
  - name: /test
    description: test tag
```

Listing 2.8: Beginning of a OAS

2.4.1 Beginning

Listing 2.8 shows the beginning of an OAS. It begins with the declaration and general information about the API where the `version` and `title` properties are mandatory. In the

servers object, API endpoint URLs can be specified for testing the API. The option to define `tags` can be used to group the specified methods. [28]

2.4.2 Methods

Further the actual API methods are defined, where for each path it is differentiated between the HTTP methods. Which method is used depends on the context, following the **C**reate **R**ead **U**pdate **D**elete (CRUD) model, whether the data is created (`POST`), read (`GET`), updated (`PUT`) or deleted (`DELETE`). Each API method has a short description and can be assigned to the defined tags. For the response it is differentiated between HTTP status codes. Every HTTP request returns a status code telling the client if the request was successful like `200 OK` or if an error occurred like `500 Internal Server Error` or `404 Not Found`. Each status code and the corresponding response is described. Here also the format of the response is specified, which for an API in most cases is JSON or also E**x**tensible **M**arkup **L**anguage (XML). `schema` describes the exact structure of the return and can include an example for easier understanding. Further response details like headers or encoding can be specified optionally. As can be seen for the status code `500` response, OAS allows to define responses which then can be referenced everywhere they are needed. [28]

```yaml
paths:
  /method1:
    get:
      summary: first method
      tags:
        - /test
      responses:
        200:
          description: Found results
          content:
            application/json:
              schema:
                properties:
                  p1:
                    type: string
                    example: Teststring
        500:
          $ref: '#/components/responses/500'
```

Listing 2.9: OAS method definition

2.4.3 Parameters

A possibility to set additional request parameters is by including them in the URL which can be used for example to specify a certain id to get data from. The corresponding syntax can be seen in Listing 2.10. The variable part of the URL is declared by a variable name wrapped in `{}`. This parameter is defined before any HTTP methods, as it is applicable for all methods in this path. The parameter definition includes if it is required or optional as well as the `schema` of the parameter, in this case only integers are allowed. There are multiple places where to put parameters. While this parameter is defined in the path, other options are placing parameters in headers or as query parameters.

```
\method2\{par}
  parameters:
    - name: par
      in: path
      required: true
      schema:
        type: integer
```

Listing 2.10: OAS parameter definition

For `POST` and `PUT` methods information can be also sent in the request body. Contrary to above described parameters, these have to be described inside the HTTP method as they are not applicable for all methods. The `schema` can also be referenced like in the response shown above for reusability.

```
put:
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/
                schemas/body1'
```

Listing 2.11: OAS request body definition

2.5 MongoDB

MongoDB is an open-source NoSQL database. In contrast to a **S**tructured **Q**uery **L**anguage (SQL) database which has more of a table-like structure, MongoDB databases consist of collections of documents which have an object-like structure that can contain nested structures. The data is stored in **B**inary **JSON** (BSON) documents, which are binary documents holding data in an extended JSON format which allows for additional data types to be stored. MongoDB collections, unlike tables in SQL databases, do not have a fixed schema, instead each document in a collection can be structured differently, allowing for more dynamic data storage. MongoDB also provides features like unique fields or indices for faster document retrieval. [24] [27] [16]

2.5.1 Aggregation

One of the most powerful features of MongoDB is aggregation. In an aggregation pipeline, multiple processing steps, called stages, can be defined, allowing combining and reshaping collections as needed. Being able to perform these processing steps in the database and not on the server reduces the number of database requests, which makes the query quicker. An aggregation starts from a collection and defines stage objects, which are executed on the data one after another, each stage passing its output to the input of the next stage. Available stages are for example a `$match` stage, only returning documents matching the given criteria or a `$lookup` stage, matching documents from another collection using the given local and foreign key. [12]

2.5.2 Mongoose

For database access from the Node.js server Mongoose is used. This module handles the low-level database connection. Collections are accessed by using Schemas. Here a document structure for every collection has to be defined, which ensures document consistency. Documents which do not correspond to the Schema cannot be saved. Mongoose however still provides a `Mixed` type that can contain any form of data like strings, arrays or objects. This provides exactly the uniform but still flexible structure needed for this use case. Further populating database queries and loading referenced data from another collection is made easier. Mongoose furthermore allows accessing the native MongoDB driver to execute more complex aggregations involving multiple collections and intermediate processing.

2.6 BIC

The **B**osch **I**oT **C**loud (BIC) provides a server structure to host applications inside the BGN. It provides hosting own applications in a container structure using so-called Buildpacks, configurable environments which provide for example a Node.js or Apache server or a Python runtime environment. A `manifest.yml` file is used to define the amount of resources available, as well as the version used and eventually needed environment variables. The BIC also provides Services, which are for example databases or a mail API, which can be bound to the hosted apps, providing additional functionality.

2.7 Regular expressions

Regular **ex**pressions (Regexs) specify a pattern to which strings can be tested for a match. This provides a far more enhanced and powerful way to extract information from strings than just searching for the occurrence of specific substrings. Next to characters itself, the number of these characters or the character group can be specified, by using special characters like `?`, `*` or `+` to specify up to one (`?`), zero or more (`*`) or one or more (`+`) occurrences. A specific number of characters can be given as `{3}`. The characters, which are allowed, can be given as `[a-z]`, specifying all lowercase characters from *a* to *z* or as a character class like `\s`, specifying a white space character. Groups, marked by `()` can be used to extract a substring from the complete match. [30]

```
/[a-z0-9._\-]+@([a-z0-9\-]+\.)+[a-z]+/
```

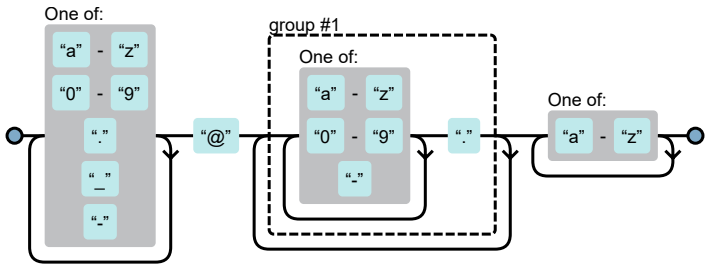

Figure 2.3: Example of a simplified email Regex

The Regex in Figure 2.3 shows an example of a regex to match valid email addresses with the corresponding railroad diagram for better visualization. It has to be noted that this is only for demonstrative purposes and by far does not cover the complete email address standard. The surrounding `//` mark a Regex in JS. The local-part before the @ must contain at least one character of the given set. The - sign thereby has to be escaped as `\-` as it is otherwise used to declare a range of characters. After the @, the domain follows with at least one domain group of the given character set. The domain parts are separated by a dot, which has to be escaped as well as a dot otherwise stands for any character. The email address is then ended by the top-level domain part.

There are of course also more advanced Regex concepts like sub patterns or conditional lookaheads, which were not covered in this short introduction, making Regexs even more powerful.

3 Implementation basics

3.1 Input validation

All input to the database has to be validated, to ensure data integrity and prevent content injection. The choice for this project was to use the *Joi* validation module. It provides a modular API which allows chaining validation arguments. Another advantage is that Joi is also available for the browser, so the same validation setup can be reused for both front and back end. An example of a *Joi* validation is shown in Listing 3.1. For an object all allowed keys can be specified with their type. The syntax is easy to read without knowing the module details: The `color` must be a string with an maximum length of 128 characters. This property is always required. The `type` is optional and has to be one of the three strings specified if given. When the `type` was given, the `with` method defines that `glass_fiber` also has to be given, which must be a number between 0 and 100. The `validate` method then validates the given data against the validation schema and returns an object containing a detailed explanation of the validation error or `undefined` as well as the value which matched the validation schema. [14]

```
1 Joi.object({
2   color: Joi.string().max(128).required(),
3   type: Joi.string().valid('granulate', 'part', 'tension rod')
4   glass_fiber: Joi.number().min(0).max(100)
5 }).with('type', 'glass_fiber').validate(data);
```

Listing 3.1: Joi validation example

3.2 Cross Site Scripting

Cross Site Scripting (XSS) is still a huge security concern for web applications. In XSS, the attacker injects malicious code into the target web application. When users visit the affected page, the injected code gets executed. The injected code enables the attacker to read cookies and other user-specific data stored in the browser for this application allowing identity theft. The malicious code can also read the user's keystrokes and send them to the attacker, giving them the possibility to collect information like passwords which the user enters on the infected site.

There are a couple of techniques to inject code into other web applications: In a *stored* attack, the attacker injects the malicious code into a user input field, like a comment on a web page. This can be done by using the HTML `<script>` tag:

```
1    <script>alert('malicious code')</script>
```

Listing 3.2: Injection syntax of JavaScript code

The comment is then stored on the server when the attacker saves his comment. Once another user accesses the page with the comments displayed, the comment with the code is automatically interpreted and executed. This behaviour can be countered in a few ways, like not directly inserting the raw user input string into the HTML source but instead using JS Methods to insert user input so that the input is treated definitely as text and cannot be accidentally interpreted as HTML. Other methods are replacing all `<>` characters with the HTML escape sequences `&lt;` and `&gt;` which are displayed as the same character to the user but not recognized as HTML Tags by the browser. Sometimes simply all `<script>` sequences are removed from the input. However this is not safe because there are multiple techniques to inject code using HTML tags aside from the shown method of using a `<script>` tag. For example the attacker can also inject an image tag with an invalid URL and specify code that should be executed if loading the image fails or inserting a link that executes a script when the user clicks on it instead of redirecting them to another page.

Another XSS method is a ***D**ocument **O**bject **M**odel (DOM)-based* attack. This type of attack can be used when a page displays a part of the URL like a specified parameter, which can be replaced by the attacker with specific code to execute. The attacker then tries to get the user into clicking the link on his own side so the user gets redirected to the page into which the attacker tries to inject his code.

The third XSS technique is called a *reflected* XSS attack. It uses the vulnerability of a server that includes a part of the user request on an error page, like a *404 Page not found*

response page. The attacker again tries to trick the user into clicking his link to send a request that contains the malicious code which is included into the response the user receives when the link is loaded. However the code is executed in the background of the error page and has access to the data and cookies stored for the targeted page from which the error response originates.

There exists an `X-XSS-Protection` header which sets behaviour if the browser detects a reflected XSS attack, which is used as follows in this project:

```
X-XSS-Protection: 1; mode=block
```

Listing 3.3: X-XSS-Protection header

This tells the browser to stop rendering the page as soon as a reflected XSS attack is detected. The problem is that this header is not supported across all browsers and provides less protection than the **C**ontent **S**ecurity **P**olicy (CSP) header, described in subsection 3.3.1, by far. It is still recommended to set this header, especially for older browsers which do not support CSP yet. [5] [7] [35]

The usage of Angular provides a big advantage in preventing the vulnerability to XSS attacks. In an Angular application, HTML, styles and URLs are sanitized by default. The HTML templates normally get pre-compiled and user data is loaded in a separate request and then injected into the the template. Thereby, user input is not directly included into HTML response by the server which could be interpreted as code by the browser. Additionally all values injected into the page are not trusted by Angular by default and therefore sanitized. Interpolated content, shown in Listing 3.4, which could include user input, is treated as pure text, so eventual `<script>` tags are only displayed but not interpreted. If HTML is generated dynamically, included in the page using `[innerHtml]`, which could include user input, this content is sanitized as well and potentially harmful content, like a `<script>` tag, is removed. A developer has to mark values specifically as trusted to bypass Angular's sanitation. This function was not used in this project to avoid potential security holes. [31] [26]

```
<p>{{interpolatedUserInput}}</p>
<p [innerHtml]="userInputAsInnerHtml"></p>
```

Listing 3.4: Injection of user input in Angular

3.3 Headers to increase user security

To increase user security, a few headers were set for the back end using the *helmet* package for Express as well as manually for the front end.

3.3.1 CSP

The CSP header provides further defence against injection attacks and XSS. Attacks are mitigated by restricting the sources from which the page is allowed to fetch resources. This prohibits loading malicious code from an attacker's server as it is not included in the list. Further all inline injection is forbidden by default which is the most frequent way for an XSS attack. However a correct configuration of the CSP header is crucial to have any effect against XSS attacks. Calzavara, Rabitti, and Bugliesi [2] found that 92% of all Alexa Top 1M websites, which were using a CSP header, had configured it in a way that made XSS attacks still possible.

The syntax of the header is as follows:

```
1 Content-Security-Policy: default-src 'self' a.com; script-src b.com;
      upgrade-insecure-requests;
```

Listing 3.5: Structure of the CSP header

All directives are separated by semicolons. Each directive begins with a directive name like `default-src` in the example and optionally a list of directive values. The most important directive names are by far the `*-src` names. These specify the sources which are allowed to be used for fetching resources of a specific type, like scripts, styles, images, etc. The `default-src` sets the fallback for all types that are not specifically declared. Not setting the `default-src` equals whitelisting every source. The list of allowed sources can be overwritten for a specific media type by setting the corresponding directive. In the given example, every kind of content can be loaded from `'self'` and `a.com` except scripts, which can only be loaded from `b.com`. The keyword `'self'` can be used to refer to the current host including the URL scheme and port number. Further keywords that can be used are `'none'` not to allow loading that type of content at all. Further there are `'unsafe-eval'` and `'unsafe-inline'` which allow the usage of functions to execute code from text and to use inline resources like `<script>` tags which should only be used if absolutely necessary as they reintroduce XSS risks.

The application of these directives for the Angular front end of the project results in:

```
Content-Security-Policy: default-src 'none'; script-src 'self'; style-
    src 'self' 'unsafe-inline'; img-src 'self'; font-src 'self'; connect
    -src https://definma-api.apps.de1.bosch-iot-cloud.com; form-action '
    none'; base-uri 'self'; frame-ancestors 'none';
```

Listing 3.6: CSP header of the Angular front end

All media types not specifically declared are not allowed to be loaded at all. Scripts, styles, images and fonts from the own host are allowed, which are necessary to display the page. The `'unsafe-inline'` keyword has to be used for styles as Angular applies its styles using `<style>` tags. However, inline styles pose a much smaller threat than inline scripts. Additionally, as already explained, Angular also has integrated input sanitation, so the risk of allowing inline styles is relatively small. Further the source of the API is specified which is needed to execute API calls and get the data to display. `form-action` is set to `'none'` as all form data is directly sent to the API using JS instead of the `action` attribute and the base **U**niform **R**esource **I**dentifier (URI) is set to `'self'`.

`frame-ancestors 'none'` tells the browser that this page may not be served inside an `<iframe>` tag. This property is also set using the `X-Frame-Options:DENY` header to support older browsers. Limiting the usage of the page inside `<iframe>` is important for user security to prevent clickjacking. In clickjacking, the attacker tricks the user into performing another action than they intended by loading the target page of the attacker inside an invisible `<iframe>`. Therefore the attacker places the button of the invisible target page directly over the fake button on his side. When the user tries to click the fake button, he rather performs the action on the attacker's target page, where he could even be personally logged in if the page features auto-login functionalities. When the page cannot be included inside an `<iframe>`, this type of attack is made impossible.

The CSP headers can theoretically be applied differently for every page served, but this is not possible in the case of an Angular single page application, because, as the name suggests, the whole application consists of one page which is served for every path and the relevant content is the computed in the client's browser.

For the back end, the header shown in Listing 3.7 is used. As the API mainly serves JSON documents and not full web pages, loading any further resources is not needed.

```
Content-Security-Policy: default-src 'none'; form-action 'none'; base-
    uri 'self'; frame-ancestors 'none';
```

Listing 3.7: Default CSP header of the back end

Apart from a **S**calable **V**ector **G**raphics (SVG) image, which uses inline styles, the only subpage, which needs special treatment, is the API documentation, which has to load the scripts, styles and images from the *swagger-ui* package. This package also makes use of inline `<style>` tags as well as `data:` URLs for images, which had to be considered in the header. However the additional risk is acceptable as all input is sanitized by the *swagger-ui* package by default. The resulting header is shown in Listing 3.8. [17] [2] [19] [6]

```
Content-Security-Policy: default-src 'none'; script-src 'self'; connect-
    src 'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:;
    form-action 'none'; base-uri 'self'; frame-ancestors 'none';
```

Listing 3.8: CSP header for the API documentation

3.3.2 DNS Prefetch Control

For higher security the **D**omain **N**ame **S**ystem (DNS) prefetching is disabled by setting the header

```
X-DNS-Prefetch-Control off
```

Listing 3.9: DNS prefetching header

With DNS prefetching enabled, the browser automatically resolves the URL hostnames found on the page to the corresponding **I**nternet **P**rotocol (IP) addresses. This reduces the load time once the user actually clicks on the link, but it can also oppose a potential privacy vulnerability. As the DNS resolution is usually not encrypted, an attacker could eavesdrop to log user activity. The resulting log is much more precise if the user's browser is doing DNS prefetching, as the attacker can pinpoint the visited site better by analysing the combination of DNS requests made because these links were included in the visited site.

Another vulnerability of DNS prefetching is the bypass of CSP. If an attacker can position any malicious code on the page but cannot send any collected information due to strict CSP settings, he can trigger prefetching by using a `<link rel="dns-prefetch"href="//information.attacker.com">` tag to send information to his server. [34] [9]

3.3.3 Removing the X-Powered-By header

Attacking a web server becomes a lot easier if the hacker knows which architecture is used. Using this information they can exploit known vulnerabilities not by the developer but by the framework itself. By default, express attaches an `X-Powered-By:Express` header, which immediately gives away the used framework. Removing this header makes attacks against the framework more difficult. [19]

3.3.4 HTTP Strict Transport Security

The HTTP Strict Transport Security header tells the browser to load this domain only over the encrypted **H**yper**t**ext **T**ransfer **P**rotocol **S**ecure (HTTPS) protocol, even if the user enters the domain with a leading `http://`. The header configuration used for both front and back end is as follows:

```
1 Strict-Transport-Security: max-age=15552000; includeSubDomains
```

Listing 3.10: HTTP Strict Transport Security header

It tells the browser to only load data from this domain over HTTPS for the next 15552000 seconds, 180 days. This should also be applied to subdomains as specified. [19] [3]

3.3.5 Further headers

```
1 X-Download-Options: noopen
```

Listing 3.11: X-Download-Options header

This header prevents execution of malicious HTML inside the served website's context. This is necessary for old versions of the Internet Explorer, where this behaviour can be achieved when the user up- or downloads HTML content.

```
X-Content-Type-Options: nosniff
```

Listing 3.12: X-Content-Type-Options header

A server usually specifies the `Content-Type` when sending data, like `text/html` or `application/javascript` so the browser knows how to handle this file. However some servers send wrong `Content-Type` headers, resulting in a failed attempt to load the website correctly. To overcome this problem, browsers do **M**ultipurpose **I**nternet **M**ail **E**xtensions (MIME) sniffing, trying to predict the correct `Content-Type`, no matter which MIME type the server specified. While this results in more successful page loads, it introduces an additional risk. Users could upload images, which are actually JS code. The browser loads this image and detects the JS code and tries to execute it, resulting in a successful XSS attack. To overcome this vulnerability, MIME sniffing can be turned off for the page by sending the header shown above. [19]

```
X-Permitted-Cross-Domain-Policies: none
```

Listing 3.13: X-Permitted-Cross-Domain-Policies header

The header in Listing 3.13 prevents loading contents from the server by Adobe products like Flash or Acrobat. These products can load content from the server even on other webpages, which the browser normally prevents for security reasons. If an Adobe product sees this header in a server response, it will refrain loading content from this server. [19]

```
Referrer-Policy: no-referrer
```

Listing 3.14: Referrer-Policy header

The browser usually sends a `Referer` header (which is spelled incorrectly in contrast to `Referrer-Policy` due to historical reasons) along with the request if the request was initiated by another page, e.g. when the user clicks a link or a page executes a redirect. The `Referer` includes the URL the user was coming from. This behaviour can have a number of insecurities, as the URL can include user-specific information, if the link the user clicked on was for example on a social media page and the URL included a username. Also servers could use the header to track users. Therefore the `Referrer-Policy` can alter the default behaviour of the browser, in which cases the `Referer` header should be included. In this case the specified value is `no-referrer` which always omits the `Referer` header as this information is not needed in the case of the project. [19] [29]

3.4 MongoDB Injection

Similar to SQL injection, hackers can inject commands into database requests to get access to data they are not allowed to see or to alter the stored data. Injection is done as shown in the following example: A server might verify user credentials using the following request:

```
1 db.users.find({username: username, password: password});
```

Listing 3.15: Database request for user credentials

If there is an entry found, which matches the given username and password, the user is given access. The problem is that username and password are often directly inserted into the JSON body of the request. An attacker now can enter valid JSON into the input fields of the webpage like `{"$gt":""}`. The server has to parse the JSON body received from the webpage's request, which results in the username not being a string, as expected, but rather an object itself. As MongoDB is queried using an object, this allows the attacker to extend commands to the query. In the given example MongoDB now searches for a username and password greater than an empty string, which will match all user entries.

Closing injection vulnerabilities can be done in multiple ways. One way is to validate all user input. Username and password for example have to be valid strings, so if the JSON parser generates an object, validation will fail. Additionally the API uses the `content-filter` module. If object keys or URL parameters starting with a `$` or a similar combination used for injection, are found, the request will be turned down. As all MongoDB operators start with a leading `$`, this removes all possible attack operators. In the defined API of this project no request key utilizes this syntax, so no actual information is lost by accident, either. [21] [4]

3.5 Software testing

Software testing is essential for high quality and bug-free code. There are multiple types of testing of which only the types used in this project are described further.

Angular provides unit testing with the integrated Jasmine framework. Unit tests are used for low-level testing close to the source code. These tests are used to test the functionalities of a single class like an Angular Service or Component. Unit tests can only determine that the class itself is working, dependencies are replaced by mocks. Interaction between classes cannot be tested by unit tests.

Interaction between Components and Services in the way a user experiences it is tested by end-to-end tests, which are also integrated into Angular using the Protractor module. These tests are used to make sure that major workflows, like a login process, are working correctly, however, they are more complex to maintain and are not used to fully test small details like unit tests do. [33] [32]

The back end is strictly speaking tested by end-to-end tests with the `mocha` and `supertest` modules. This is due to the fact that the whole server and database are running for every test. However as this is a ReST API and every request is self contained testing is done in a unit test style: Every API method is still tested alone without any other method involved.

4 Implementation

4.1 Database

The database is the heart of the project. As all data should only be stored on this database in the future, it had to be ensured that data cannot be lost or corrupted in any way.

4.1.1 Database model

A study to find a suited database model for the given data to store was already conducted in a preceding work [10], the result can be seen in Figure 4.1. It was found that a NoSQL database using templates was fitted best for the currently available data, while considering future use cases. The templates provide a way to maintain a fixed structure to guarantee uniform data while still allowing modifications over time, due to changing requirements.

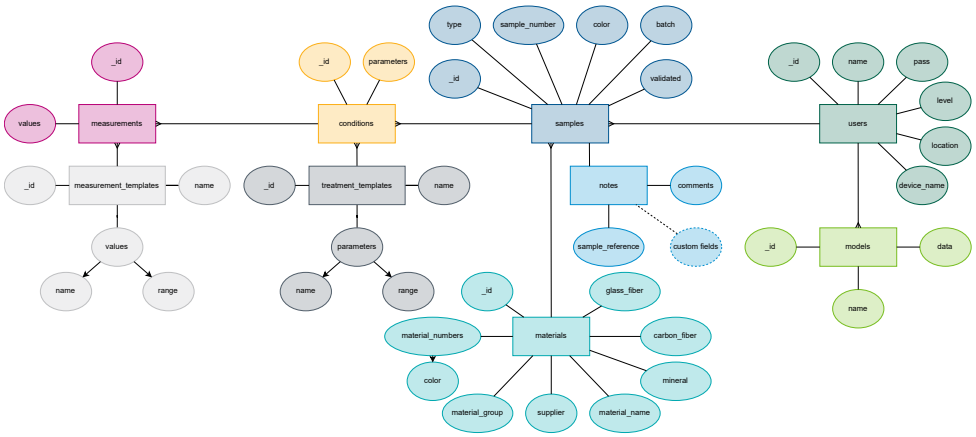

Figure 4.1: Database model from the previous work [10]

Figure 4.2 shows the final implemented version of the database model. The `templates` now are versioned, as in case a `template` is changed, it is not feasible to change all documents relying on this `template` as well. The `first_id` points to the first version of this template to

be able to group all versions of a `template`. As there might be further use cases, material-specific properties are now also definable using a template. Also `group` and `supplier` were put in separate collections to avoid grouping materials of the same `supplier` into multiple groups due to misspellings.

The `condition` was now put into the sample collection as there is only one `condition` per `sample` and there is no need for an extra collection, as assumed in the previous version. Also the `custom_fields` keys are now kept in an extra `note_fields` collection to be able to use this information quickly for providing suggestions in the user dialogue. Further a changelog collection was introduced to keep track of all changes and additions to the database.

The former `models` collection was now renamed to `model_files`, as it only stores the binary model data, while the new `model` collections is used for configuring the `predictions` in the **U**ser **I**nterface (UI). There are multiple prediction groups, like *VN* or *moisture content*, which can each contain multiple model entries, because, as previously explained, different materials need different prediction models. Each model entry contains next to a `name` a `label` to describe in which unit the prediction is returned and the `url` which specifies the API endpoint of the Python prediction container.

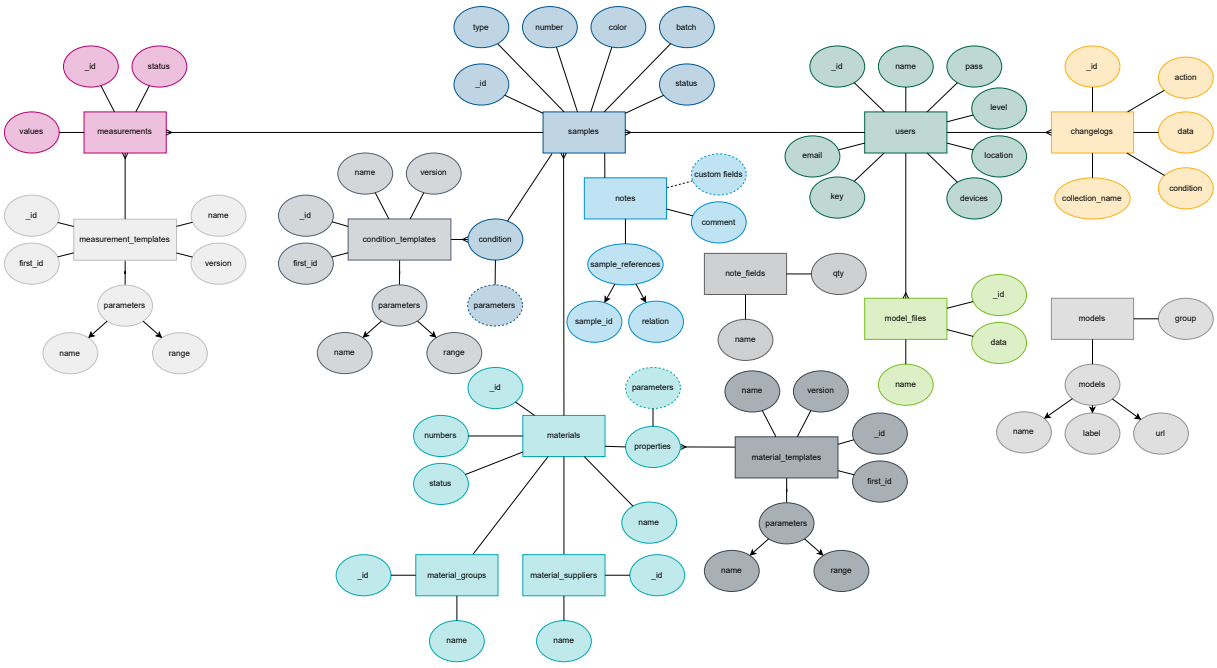

Figure 4.2: Final database model

4.1.2 MongoDB

As the database engine, MongoDB is used. On the BIC the engine is provided by a service bound to the app. Binding the service directly to the app secures the database against unauthorized access. The database cannot be accessed directly but only by the bound app. Credentials are also provided as environment variables for the app, so no database passwords have to be encrypted or stored in the codebase.

4.2 General server structure

On startup, the server executes a number of tasks. First of all it connects to the database using the right connection string, depending on whether the server starts in a development, production or test environment. Then the server middleware is set up. This includes the headers discussed in section 3.3 and escaping against injection attacks as well as compression for gzipping the response, and parsing the JSON request body. Requests, which result in errors during middleware execution are automatically returned to the client. This includes requests with an invalid JSON body or a request containing possible injection code. As every route apart from the root method and the API documentation requires database access, all requests are automatically returned with a `500 Internal Server Error` if no active database connection is found. Furthermore the authentication provided by the user is checked and information, whether Basic Auth or an API key were used, and which access level the user belongs to, is attached to the request data that is passed on to the actual API routes. As different routes require different levels and authentication methods or even no authentication at all, requests are not returned as unauthorized at this point as this has to be done on a per route level.

Then the actual API routes are included, followed by static files which have to be served. Additionally the API documentation is put together into one document and validated to detect any OAS syntax errors. The validated document is hosted using the `swagger-ui` package as already described.

Finally routes are described to catch all remaining requests to return a `404 Not Found` response, as well as a route catching all propagated errors from the preceding routes to return an `500 Internal Server Error`. This concludes the setup, whereafter the server is started.

4.3 Authentication and user levels

The server supports two authentication methods: HTTP Basic Authentication and specifying an API key as a query parameter.

The API key is a less secure method of authentication. When a new user is created, a unique key is generated as well, and stored in plain text in the database. This is necessary as the key has to be returned to the user, whenever they request it. As the key is also part of the URL by appending it using the format `&key=xxxxx`, this key could be stolen more easily. The back end by default uses HTTPS, so the query parameters should be encrypted, but the full URL, including the query parameters, sometimes gets stored, for example, in server logs. If an attacker gains access to these logs, they get access to the API keys of the logged requests as well.

Basic Authentication in contrast provides more security as the credentials are sent in the `Authorization` header. The user password is stored in the database in its hashed form using the *bcrypt* algorithm and only the hash of the stored password and the hash of the password provided for authentication are compared.

Each user account belongs to one of four levels: *read*, *write*, *dev* or *admin*. Table 4.1 shows the key differences in which rights each level possesses. As good machine learning models can only be created with a big database of high quality data, this data should not be given away too easily. A *read* user therefore can view the sample details and get an overview of the recorded data, but cannot access the infrared spectra, as these are the main source to train machine learning models on. Users, who contribute measurements and spectra, get a *write* level. They can create new samples and edit their own samples, as well as delete them. However, they can only upload, but not download spectra. Regarding viewing samples, they have the same rights to view the main properties of all samples available. The *dev* level is given to developers creating machine learning models, as well as to persons who need to maintain the database. They can edit and delete samples no matter by which user the samples were created. They have the ability to directly download the current sample view in **C**omma-**S**eparated **V**alues (CSV) or JSON format to process it further using own scripts. In contrast to *read* and *write* users, they also have the ability to view deleted samples and measurements and restore them if necessary. Users with the *dev* level can also create and edit templates for *materials*, *measurements* and *conditions*, which were described in subsection 4.1.1. Last but not least these users can also access the spectra measurements as this data is needed for their work. The *admin* level is basically equal to the *dev* level with the addition of user management, being able to add, edit and delete users, except from changing the password.

	read	write	dev	admin
read sample data	yes	yes	yes	yes
add samples/edit own	no	yes	yes	yes
read spectral data	no	no	yes	yes
edit other's data	no	no	yes	yes
maintain templates	no	no	yes	yes
edit users	no	no	no	yes

Table 4.1: Permissions for the user levels

The authentication and level needed to be able to access certain API methods is described for each path individually in the OAS. Generally, the API key authorization is only possible for `GET` methods of *samples*, *materials*, *measurements* and *models* as reading this data is required in the developed Python scripts for machine learning. Here appending the key to the URL is easier as it avoids storing the user credentials itself in the code. For any kind of write access to the database and read access to other routes, Basic Authentication is needed.

4.4 Routes

The routes are structured similarly to the database. Next to a few root routes, there are routes for `/sample`, `/material`, `/measurement`, `/template` `/model` and `/user`

4.4.1 `/samples` route

The `/samples` route returns the samples including all information which references the sample or is referenced by the sample. Which data should be returned is specified by query parameters:

`status[]` specifies which status the returned samples might have. The available statuses are `new` and `validated`. *dev* and *admin* users are also allowed to request samples with the status `deleted`

`from-id`, `to-page`, `page-size` are used for pagination, as implemented in the samples overview in the UI. The need of three parameters for efficient pagination is due to the way MongoDB can handle these requests. In order to return the nth page with x results, MongoDB has to generate the first $n \cdot x$ results just to then discard the first $(n-1) \cdot x$ results. As this way of requesting data becomes really slow for complex queries and high page numbers, the `from-id` parameter is included in the request as well, referencing the first sample of the page the user is currently viewing. `to-page` also does not request an absolute page number, but rather a positive or negative number specifying the number of pages to go forth or back relative to the referenced sample. If the user is for example on page 40 with 30 elements and wants to see the next page, the database can start looking from the referenced sample and only has to discard 30 samples instead of 1200, resulting in a much better performance.

`sort` specifies the column to sort the samples, followed by `-asc` or `-desc`, depending on whether the samples should be sorted ascending or descending. Sorting is possible for all keys with simple values like numbers or strings. Sorting by properties like *condition* or *notes* is not supported as these properties have a complex and non-uniform structure.

`fields[]` has to be set to indicate the fields included in the response. Possible values are of course the sample properties defined in the database like `number` or `color`, but also referenced values from the *measurements* or *materials* collection, indicated by a dot like `material.name` or `measurements.spectrum.device`.

`filters[]` allows for filtering the samples included in the results. Every rule of this array is an URIcomponent encoded object specifying a filter rule. The object contains the field to apply the rule to, which has the same allowed values as `fields`. Additionally a `mode` property has to be specified for the comparison operator, the possible values are listed in Table 4.2. The values, to which the field should be compared to, have to be listed in the `values[]` array of the filter rule object. It has to be noted that for all modes where only a single value is required, which are all modes except `in` and `nin`, all values except the first one in the array are ignored.

`output` can be set to `json`, `flatten` or `csv`. This parameter controls the output format. In contrast to the default `json` mode, which returns the sample data in a normal nested object structure, `flatten` transforms the object into an object without nested sub-objects. For example `{"material":{"name":"Ultramid","color":"black"}}` gets transformed

mode	description	UI sign
eq	field is equal to value	$=$
ne	field is not equal to value	$\neq$
lt	field is lower than value	$<$
lte	field is lower than or equal to value	$\leq$
gt	field is greater than value	$>$
gte	field is greater than or equal to value	$\geq$
stringin	field contains value	$\supseteq$
in	field is one of the values	$\in$
nin	field is not one of the values	$\notin$

Table 4.2: Available filter modes

into `{"material.name":"Ultramid","material.color":"black"}`. Further the spectra are transformed from a single two-dimensional array like `[[label,value]]` into two one-dimensional arrays, one containing all labels and the other one containing all values. The resulting flattened object provides an easier import into *Pandas Data Frames*, the data format used in Python for machine learning. The `csv` option uses the same flattened object and returns it in CSV format for easy use in Excel or similar spreadsheet programs. It has to be noted, that here the spectrum itself is omitted, as the amount of data could not be displayed in a neat way.

```
1 /samples?status[]=new&status[]=validated
2 &from-id=5f2e64438d1c020f8cda7b09&to-page=1&page-size=25
3 &sort=added-asc
4 &fields[]=number&fields[]=type&fields[]=color&fields[]=batch &fields[]=
      status&fields[]=added&fields[]=material_id&fields[]=_id&fields[]=
      user_id
5 &filters[]=%7B%22mode%22%3A%22eq%22%2C%22field%22%3A%22material.name%22%
      2C%22values%22%3A%5B%22Ultramid%20A3EG6%22%5D%7D
6 &filters[]=%7B%22mode%22%3A%22stringin%22%2C%22field%22%3A%22color%22%2C
      %22values%22%3A%5B%22black%22%5D%7D
```

Listing 4.1: Example of a `/samples` request

Listing 4.1 shows an example for a `/samples` request. The new and validated samples are requested for the next page with 25 items, starting from a referenced `_id`. The fields which should be returned are defined as well as two filter rules: `{"mode":"eq"`

`,"field":"material.name","values":["Ultramid A3EG6"]}` specifies that the `material name` must be *Ultramid A3EG6*, whereas `{"mode":"stringin","field":"color","values":["black"]}` additionally specifies that the color of all samples must contain the string *black*.

Fetching the stored sample data with sorting, filtering and displaying the right fields in an efficient way proved quite a challenge. In this case the key to create an efficient MongoDB aggregation is to only handle the minimum amount of data along the aggregation stages. Key operations therefore are pagination and filtering as they reduce the size of the dataset given to the upcoming stages. The whole process is displayed in Figure 4.3

After validating user credentials and request parameters, the key priority is getting the samples sorted by the specified key in as few stages as possible. Depending on whether the result should be sorted by a `measurement` key or another key, the aggregation is applied to either the `measurements` collection or the `samples` collection. Especially the spectrum measurements are very large in size as a single **D**ata **P**oint **T**able (DPT) file has a size of 35 KB. The `measurements` are filtered by template and sorted accordingly. Then the referenced `sample` is joined and the data restructured to have the same structure as a `sample`, so all following steps can be applied equally, independent of the starting collection. Now that the `sample` information was joined, samples with the wrong `status` can also be filtered out.

If the starting collection is the `samples` collection, the `status` can be filtered right away. If the sorting key is in the `samples` collection, it can already be applied. The gained samples, no matter whether from the `samples` or `measurements` collection, can then be filtered by the `sample` filter rules given, to further reduce data size. If the result is to be filtered by a material key, the `material` data and eventually `material group` or `naterial supplier` have to be added. The results are always filtered right away to reduce the working data size.

At this point in the aggregation pipeline the data is definitely sorted and could be paginated, as going down from possibly a few thousand samples to less than 100 that fit on a page drastically reduces data size. However, in pagination view the user needs the number of total items displayed. To count the number of samples, all filters have to be applied. Therefore measurements needed for the filters are joined first and filters are applied respectively. Then all filters are applied and the number of resulting samples cannot change anymore. The samples are now counted and then paginated as previously described. The sample number however can only be determined if no `from-id` is specified. For efficient pagination, samples with a smaller sort property are filtered out in the first aggregation stage. This limitation does not pose a problem as the first page, which does not require a `from-id` is displayed every time a filter or sort parameter changes where the sample number

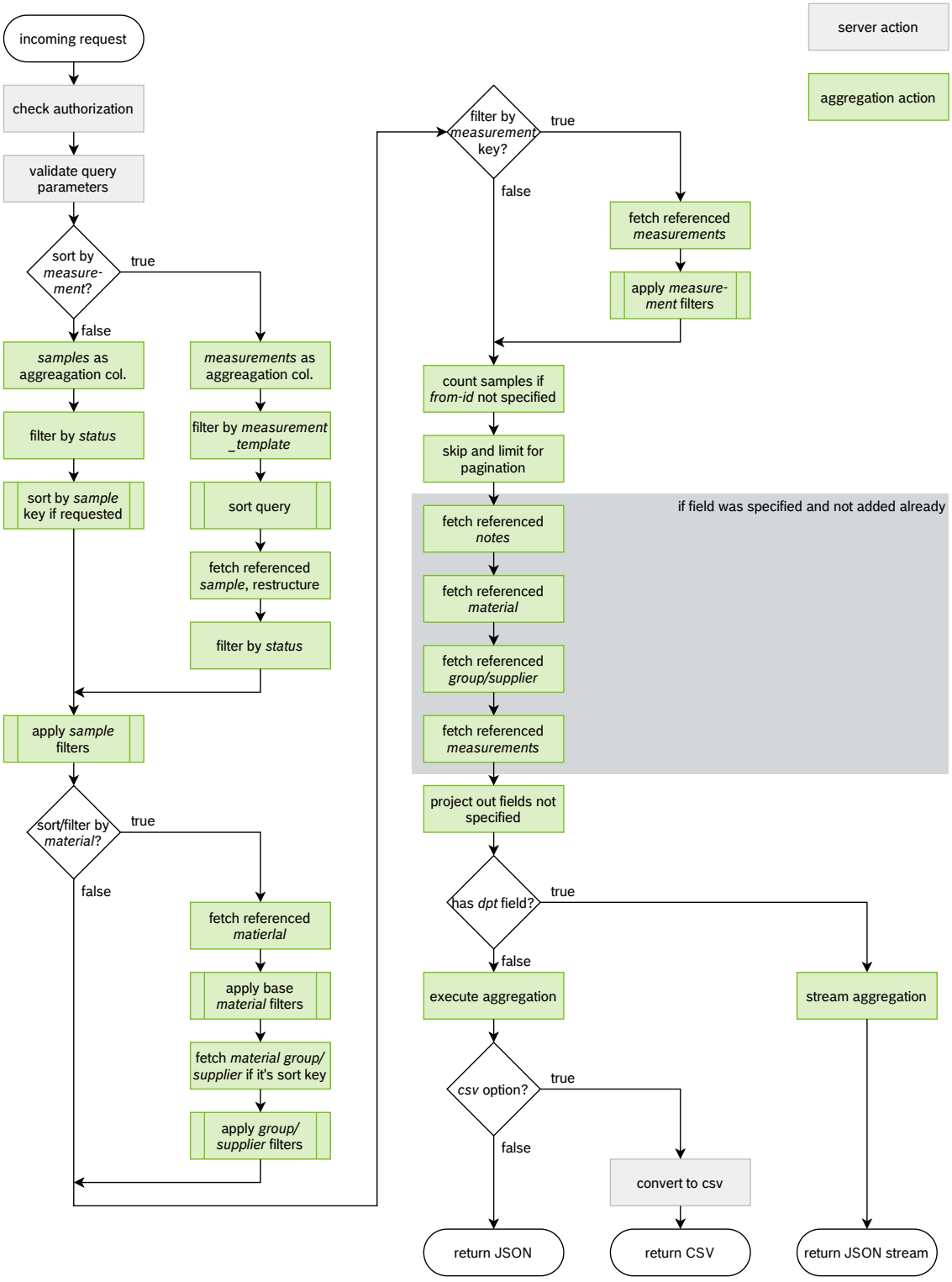

Figure 4.3: `/samples` aggregation build-up

can be determined. This number is stored in the UI, when the user goes through further pages.

After pagination, additional specified `fields` are joined to the dataset, if they were not required for filtering or sorting. This includes `notes`, which cannot be used for sorting or filtering, as well as `materials`, including `groups` and `suppliers`, and `measurements`, as they were not joined earlier if these values were not required, which keeps working data size down. At the end, fields, that were required for joining, or when excluding them earlier had slowed down the aggregation, are projected out.

The whole aggregation, that was built up on the server, is then executed in the database. As previously explained, the resulting data size is a lot bigger if spectra are included, so in this case the aggregation result is directly streamed to the client, resulting in smaller server memory load. Otherwise the aggregation is executed normally and either returned directly, or converted into a CSV file if specified. The generated aggregation pipeline for the example URL described above can be seen in Listing 4.2.

```
samples.aggregate([
  { '$match': { '$and': [ { '$or': [ { status: 'new' }, { status: '
      validated' } ] } ] } },
  { '$match': { '$or': [ { _id: { '$gt': ObjectId('5
      f2e64438d1c020f8cda7b09') } }, { '$and': [ { _id: ObjectId('5
      f2e64438d1c020f8cda7b09') }, { _id: { '$gte': ObjectId('5
      f2e64438d1c020f8cda7b09') } } ] } ] } },
  { '$sort': { _id: 1 } },
  { '$match': { '$and': [ { color: { '$in': [ /black/ ] } } ] } },
  { '$lookup': { from: 'materials', localField: 'material_id',
      foreignField: '_id', as: 'material' } },
  { '$addFields': { material: { '$arrayElemAt': [ '$material', 0 ] } } }
      ,
  { '$match': { '$and': [ { 'material.name': { '$eq': 'Ultramid A3EG6' }
       } ] } },
  { '$skip': 25 },
  { '$limit': 25 },
  { '$project': { number: true, type: true, color: true, batch: true,
      status: true, added: true, material_id: true, _id: true, user_id:
      true } }
]);
```

Listing 4.2: Example of a generated `/samples` aggregation pipeline

4.4.2 Further `/sample` routes

The other `/sample` methods have less options and therefore less processing aside from validation and data formatting.

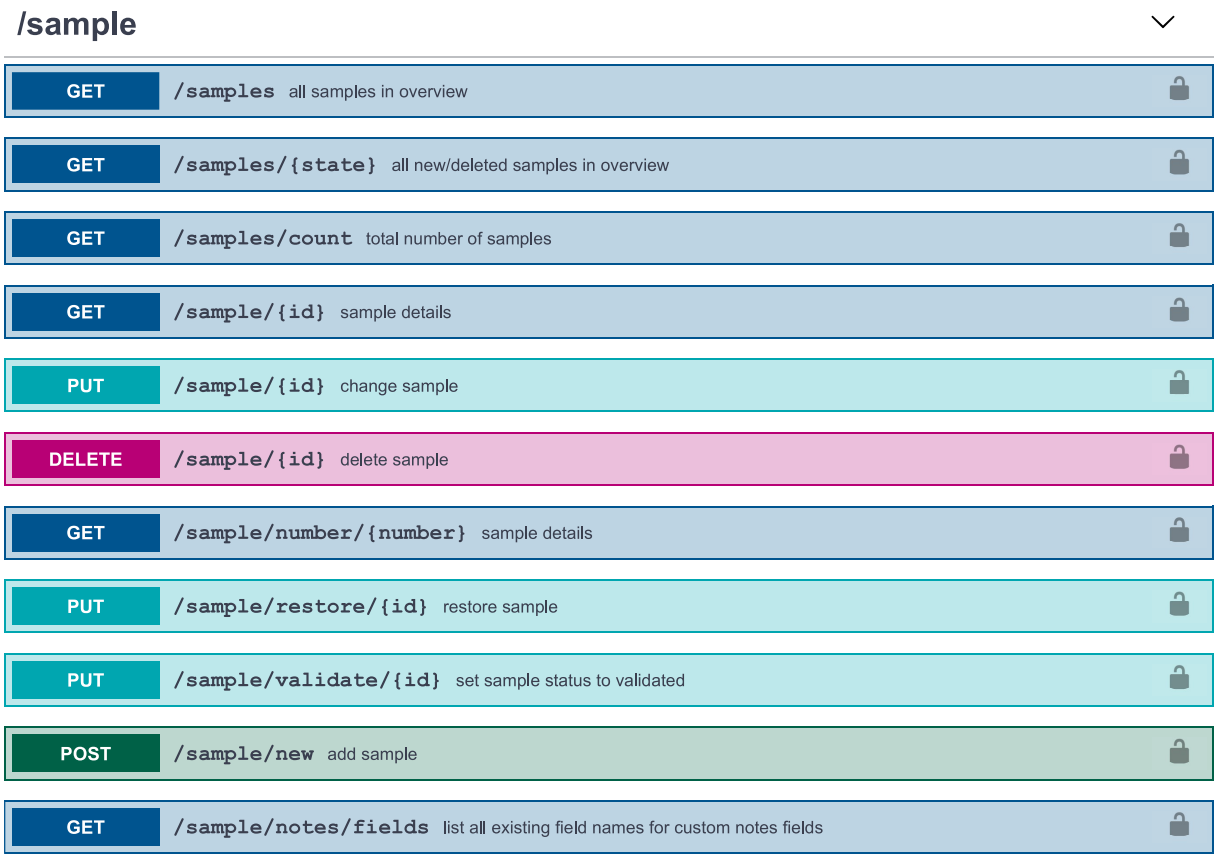

Figure 4.4: All `/sample` routes

`/samples/{state}` returns the `samples` collection documents filtered by `status` specified in the `state` parameter. Here the documents are not joined with other collections but instead only the fields directly included in the collection are returned. The only exception is the `added` field, which is generated from the `_id`.

`/samples/count` only returns the number of samples currently in the database, irrespective of the document's `status`.

`/sample/{id}` returns all details of a single sample in the GET method. To this end the Mongoose `populate` feature is used. Next to the sample data, the `notes`, `material` and all `measurements` are included in the response, which can be directly utilized for the sample

details dialogue in the UI. If the `_id` of the sample is not known, the sample details can also be requested by referencing the `sample number` using the `/sample/number/{number}` route.

In the PUT method, only the main `sample` properties can be edited. Every sample can be edited by the *write* user who created it, as well as every *dev* and *admin* user. Apart from validating all input properties for a valid type and range using Joi, the `material_id` is also checked for a valid reference to an existing material. For the given condition the `condition_template` has to be checked. The template has to be the latest version of this specific template, if the condition was newly specified. If the sample already had an old version of condition template, it is allowed to keep this version. Then all condition parameters are validated to the dynamically defined constraints of the template. For the `notes`, the `note_fields` are updated accordingly. If there have been actually any changes compared to the existing `sample` values, the `status` of the sample is set again to *new*, then the changes are saved in the database.

The DELETE method again requires the user either to be the owner or to have *dev* or *admin* level to delete the sample specified by the `id`. If this is the case, the sample `status` is set to *deleted* as well as the `status` of all `measurements` belonging to this sample.

`/sample/restore/{id}` allows *dev* and *admin* users to restore deleted samples. By only using the `status` to flag samples as *deleted*, instead of actually removing them from the database, this is easily possible. It avoids losing data which was accidentally deleted, or keeping data in the database, which is not a valid sample, but could still be needed in the future.

`/sample/validate/{id}` works exactly like the `restore` method described above, but sets the status to *validated* instead of *deleted*. This method is only allowed for *dev* and *admin* users.

`/sample/new` does pretty much the same as the PUT `/sample{id}` method. The only difference is that all sample fields have to be included in the request body and not only the fields which should be changed.

`/sample/notes/fields` returns the data from the `note _fields` collection. This includes the names of all `custom_fields` in the `sample notes` as well as the quantity, in how many `samples` this key was used. This information is used for autocompletion when samples are added or edited, resulting in quicker data entry and reducing the possibility, that two users enter

a similar key with the same meaning, unaware that the key they are looking for already exists. Having the same information in multiple keys would make data processing a lot harder.

4.4.3 `/material` routes

The most `/material` routes function similar to the corresponding `/sample` routes. They only join the `material_groups` and `material_suppliers` data with the main `materials`. These two properties are excluded into own collections as the same values appear in multiple documents. In this way, the entries can be directly extracted, as done in the `/material/groups` and `/material/suppliers` routes, to provide autocompletion in the UI. This avoids spelling mistakes and allows easy renaming in case a `group` or `supplier` name should be changed. However `group` and `supplier` are sent to the API in normal string form and resolved by the server if the name already exists, otherwise a new entry is created automatically. The `material name` of the material must be unique, as it is used for autocompletion of the whole material, if it already exists in the database. If two materials had the same name, the user would not be able to definitely pick the right material only by the suggested names.

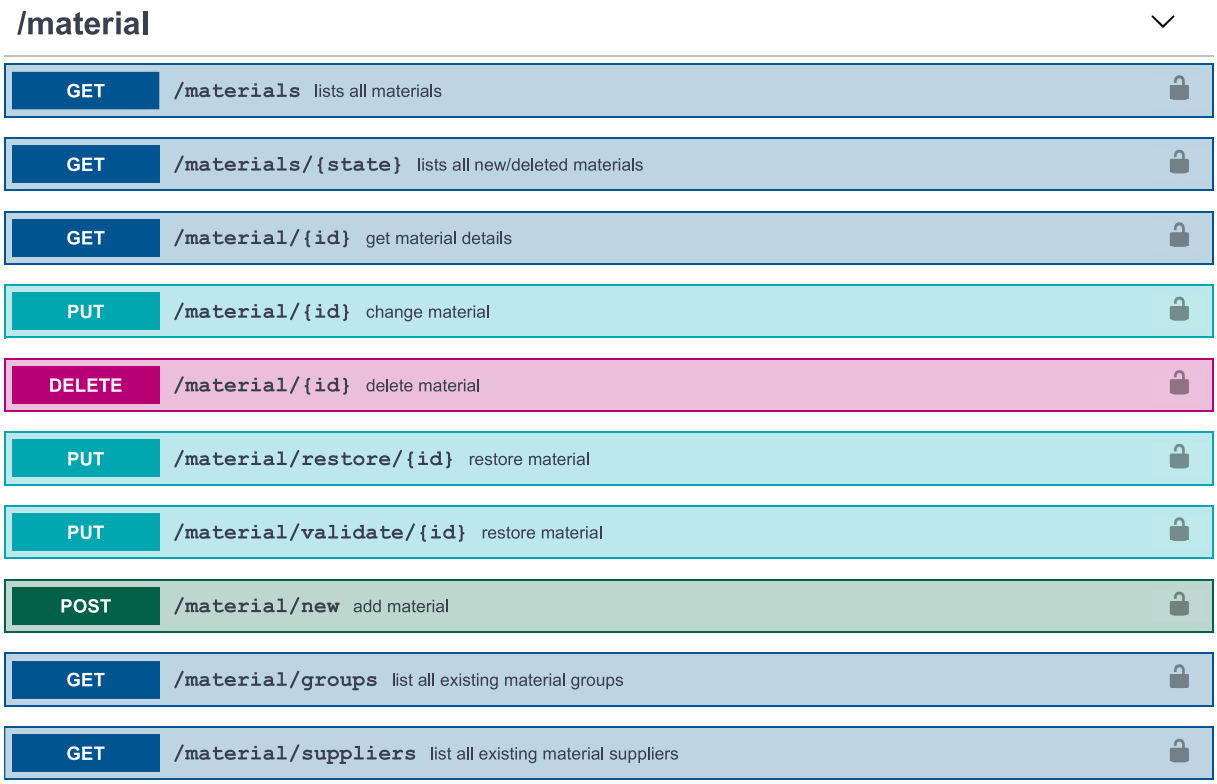

Figure 4.5: All `/material` routes

4.4.4 `/measurement` routes

The `/measurement` routes implement the same functionality as the aforementioned routes in a reduced set. Listing all routes or all routes with a specific `state` is not implemented as this view is not needed anywhere in the application. More importantly, listing all measurements includes listing all spectra, which would result in a huge amount of data. Measurements with the *deleted* status are also only accessible by *dev* and *admin* users.

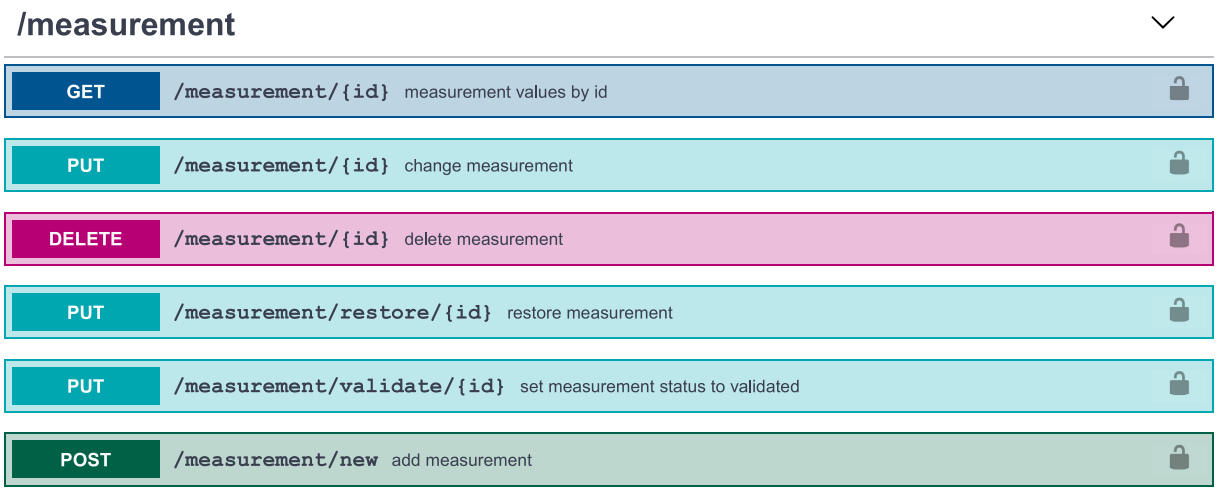

Figure 4.6: All `/measurement` routes

4.4.5 `/template` routes

The template routes allow *dev* and *admin* users to create and modify `measurement_`, `material_` and `condition_templates`. These templates allow adapting the database to different use cases by dynamically defining allowed object structures. The available templates of one of the three available collections can be listed and edited. To allow backwards compatibility to existing documents, editing a template does not simply change the template entry. Instead a new document, describing the new template details, is created and given a new version.

A `template` document, as shown in Listing 4.3 contains next to `_id` and `version` a `first_id` parameter, allowing to group all versions of a template as the `name` property could change over time. The actual parameters, which can be specified by the user in the `measurement`, `material` or `condition` document are defined in the `parameters` array. The `name` is used for display as well as the key in the document. The `range` properties can be specified for validation of the parameter. In the given example, minimum and maximum of the value are defined, implicitly also defining that the value must be a number. These properties

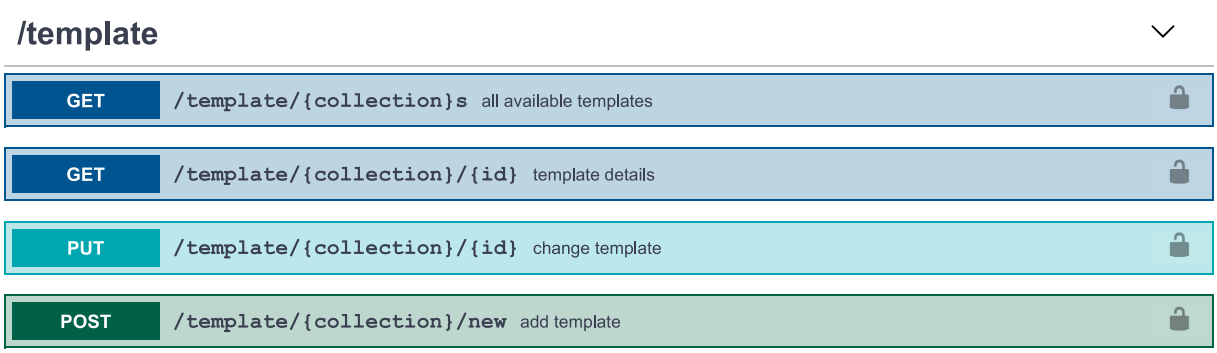

Figure 4.7: All `/template` routes

can also be set separately to limit the range only in one direction. There are two more possible properties available which are not shown. These are `values`, which specifies an array of allowed values, providing to the user a drop down menu to select one value in the UI and the property `type: array`, which allows setting an array instead of a simple value, which is currently only used for spectra.

```
{
  "_id": "300000000000000000000002",
  "first_id": "300000000000000000000002",
  "name": "moisture content",
  "version": 1,
  "parameters": [{
    "name": "weight %",
    "range": {
      "min": 0,
      "max": 1.5
    }
  }]
}
```

Listing 4.3: Example of a `/template` object

4.4.6 `/model` routes

The `/model` routes are used for machine learning model management. The first three routes shown in Figure 4.8 are used to access the data from the `models` collection. `/model/groups` just returns all documents from the collection, listing all stored groups with the corresponding models. `/model/{group}` allows to add a new model with `name`, `label` and `url` to the given `group`. The model `name` has to be unique, which results in an old entry being replaced when a new entry with the same name is stored. By specifying `group` and `name` in

the `/model/{group}/{name}` route, a certain model can also be deleted again from the list. As the stored data is not that critical, data is actually deleted in comparison to `samples` or `measurements`, where only the status is set to *deleted*.

The `/model/file/{name}` routes are used to store the binary machine learning model files. As the containers in the BIC itself lose all state when they are restarted, the model files either have to be uploaded after every start or need to be pushed into the BIC together with the source code. Storing them along with the sample data in the MongoDB proved as a better solution, making it easier to change model files and share these files between developers and containers.

The three available routes provide basic file access. The GET method returns the binary data stored under the given name, the POST method stores new data under the given name and overwrites any data previously saved with this name, and finally the DELETE method allows deleting data.

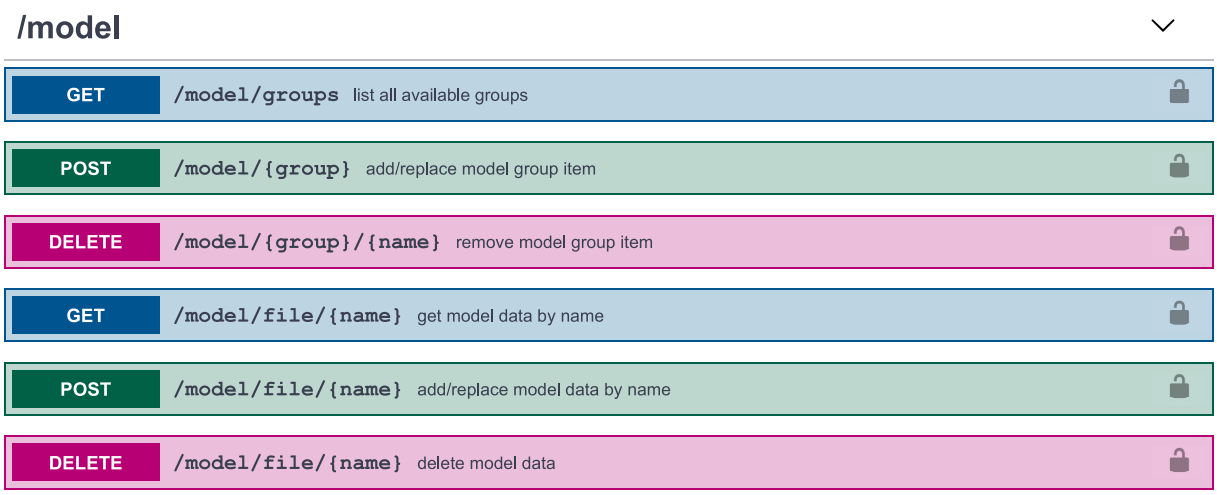

Figure 4.8: All `/model` routes

4.4.7 `/user` routes

The `/user` routes allow the user to change their account settings, whereas the `/user/{name}` routes allow *admin* users to change the same settings for any account. Of course the `level` can only changed by an *admin* and not the user themselves. An important case is the `email` change: As `username` and `email` are all that is required to request a new password using the *forgot password* function, an attacker could try to change the `email` to send a new password to themselves. Therefore an email is sent to the old email address notifying the user to which address the `email` was changed so the user can report unauthorized activity.

The `username` can be also changed after account creation, but of course has to be available as the `username` is a unique field.

The `/users` route allows the *admin* to get an overview of all users and their account details. They can create new users using the `/user/new` route, where the same rules apply as described for editing users. The *admin* can assign an initial password, which is useful for user accounts created for development. When creating accounts for other users, the *admin* can either pass on the initial password or simply assign a random password. In this case the user can utilize the *forgot password* function, as long as the *admin* assigned their correct email address. This way, no password has to be passed at all.

The `/user/passreset` method provides the *forgot password* function and takes `username` and `email` as identification credentials. It then generates a new password and stores it. This password is sent to the user utilizing the BIC mail service.

Finally the `/user/key` method returns the API key of the authorized user. The key has to be requested by the UI when download links are generated. That way the user only has to paste the link into the desired application.

Figure 4.9: All `/user` routes

4.4.8 Other routes

`/` is only used to determine if the API is working normally, in which case it returns a `{"status":"API server up and running!"}` object.

`/authorized` is a route used in the UI to determine whether the given credentials are correct. In this case the method returns the given authorization `method`, either *basic* or *key*, the user `level` as well as the `user_id`. If the given credentials were incorrect, an HTTP status code of 401 together with an `{"status":"Unauthorized"}` response is returned.

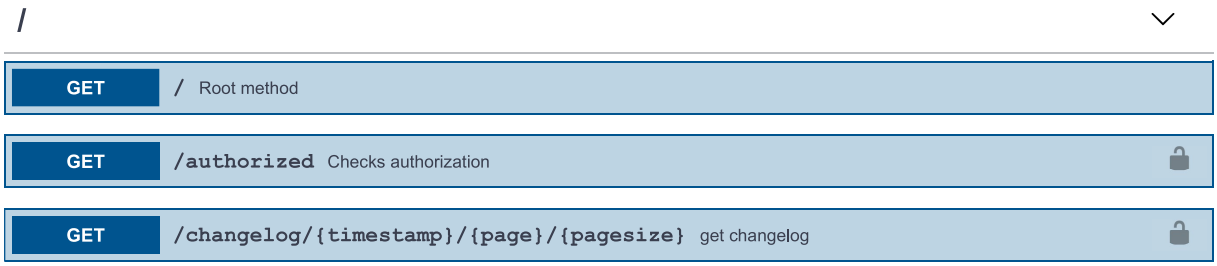

Figure 4.10: All other routes

`/changelog/{timestamp}/{page}/{pagesize}` returns the changelog data from the given timestamp sorted descending by date with the same paging mechanism as used in the `/samples` route.

All actions, changing any data on the database, are logged into the changelog collection. The logging function has to be called as part of the Mongoose middleware with only the `request` object as an argument, like shown in Listing 4.4, all other information is taken from the preceding Mongoose function. Only when creating new entries with the `.save()` method, log creation has to be called in a separate function call.

```
1 MeasurementModel.findByIdAndUpdate(id, data).log(req).lean().exec()
```

Listing 4.4: Mongoose database call with logging

The data saved for every change includes the properties shown in Listing 4.5. It shows the `action` in which response the change was made, in this case a call to the `/material/new` route to create a new material, as well as the collection, to which this change was made, in this case the `material_groups` collection. By the stored `conditions` and `data`, it can be seen, that in creating the new material, the group *PBT* was newly saved as it has not existed yet.

```
{
  "_id" : ObjectId("5f2e63118d1c020f8cda6a08"),
  "action" : "POST /material/new",
  "conditions" : {
    "name" : "PBT"
  },
  "data" : {
    "name" : "PBT"
  },
  "user_id" : ObjectId("000000000000000000000003"),
  "collection_name" : "material_groups"
}
```

Listing 4.5: Changelog entry

4.5 Data import

An essential task was to import the existing data from spreadsheets and spectrum DPT files into the newly created database. Therefore, all metadata tables were converted into CSV and all DPT files were copied into one location. The import consists of three stages, first importing `materials`, then the `sample` data from the metadata tables and finally the DPT files. All stages can be executed independently.

The `material` import iterated through all metadata tables. Each table in CSV format was loaded into the script at first. As the table headers of all files did not have all the same properties, the columns first had to be brought into a uniform format. The first step was to remove all column name spaces and convert the whole string to lowercase. In a second step, the keys were converted using a manual maintained object, which mapped keys with wrong names or in the wrong language to the uniform key name. As some keys like `color` or `batch` are not required, these columns could be substituted with empty values, if the row did not exist in the file. Further some custom compromise replacements were made, for example the `material group` like *POM* was also used as the required `material name`, if not given. Rows which still did not possess all necessary properties were removed and logged for further manual error handling.

Then all materials were extracted from the metadata by iterating through all sample entries. As the `material name` is required to be unique, it was used to differentiate between materials, which already occurred at least once, and completely new materials. For existing

materials, only color and material number were checked, and, if they were new, eventually added to the list of available colors and material numbers available for this material.

For new materials, some properties could be stored directly, like material name or supplier. But to find all material numbers as defined in the *Bosch Normmaster*, which differ between material colors, this information had to be retrieved separately. Therefore the *Bosch Normmaster* web page was downloaded and scanned for the given material number. If the number could be found, the corresponding **P**ortable **D**ocument **F**ormat (PDF) file was downloaded and all available material numbers and corresponding colors extracted. Using this information, the given color and and material number could be cross-checked for validity and faulty sample entries could be reported.

In the metadata files, the reinforcing material was also stored in one field and the type of reinforcing material specified by an abbreviation. For the database, this field was split into the properties carbon_fiber, glass_fiber and mineral. Once all materials were fully extracted, they were uploaded to the database.

To import the sample metadata, the tables in CSV format were imported again in the first step. Then all valid rows were processed. Some properties like number, type or batch could be imported directly. From the given material name, the corresponding _id of the material, that was previously created, had to be found. Deriving the custom_fields from the input data proved to be more difficult, as this field did not exist, but rather all information was put in the comment field. Therefore all comment fields of the input data were manually evaluated and rules for suitable data were defined. Examples of different rules are listed in Listing 4.6. Each rule consists of a docKey, which has to be found in the comments field so this rule is applied. The dbKey defines the translated version, which should be used as a custom_fields key in the database. The Regex matches the part of the comment to use where the first group always is the custom_fields value inserted into the database. While the most rules belong to the custom_fields, sample_references and vn measurements could also sometimes be found in the comment. They were also matched and stored for later use. The part of the comment, which matched the Regex, was removed. The remaining part of the comment, which could not be matched by any rule was then returned to be put in the comments field in the database.

In a last processing step, color and material number were completed, where possible. If one of these properties was missing, the other field was supplemented using the *Bosch Normmaster* data. For subsamples, where *moisture content* or ***V**iscosity **N**umber (VN)* were measured, these properties could also be supplemented from the main sample. Then the processed sample data was saved. As mentioned, for some samples, the metadata

rows also contained measured *moisture content* including the *standard deviation* or *VN* as well as *reinforcement material content*. These values were also extracted and saved in the `measurements` collection, referencing the `sample` by its `_id`.

```
[
  {docKey: 'Berstdruck', dbKey: 'bursting pressure', regex: /Berstdruck:
       (.*?bar);/, category: 'customField'},
  {docKey: 'granulate zu', dbKey: 'granulate to', regex: /granulate zu.*
       ( S* d+)/, category: 'reference'},
  {docKey: 'VZ:', dbKey: 'vn', regex: /VZ: ([0-9.,]+) mL /g[;]?/,
      category: 'vn'}
]
```

Listing 4.6: Examples of the defined `comments` rules for data extraction

The last step was the import of the spectrum files. Therefore all spectrum files were checked and uploaded, if possible. This started by bringing the DPT file names, which were not named according to the naming guidelines, in the uniform format. These wrong file names were filtered out by the Regex in Figure 4.11. Only the groups containing the sample location and number were kept, additional white spaces or `_JDX` fragments from conversion were removed. Spectrum names, which did not match the Regex, were only checked for the `_JDX` ending, to remove it if it exists.

```
/^(Bj[FT]?)\s?([a-z0-9_]*)_JDX.DPT/
```

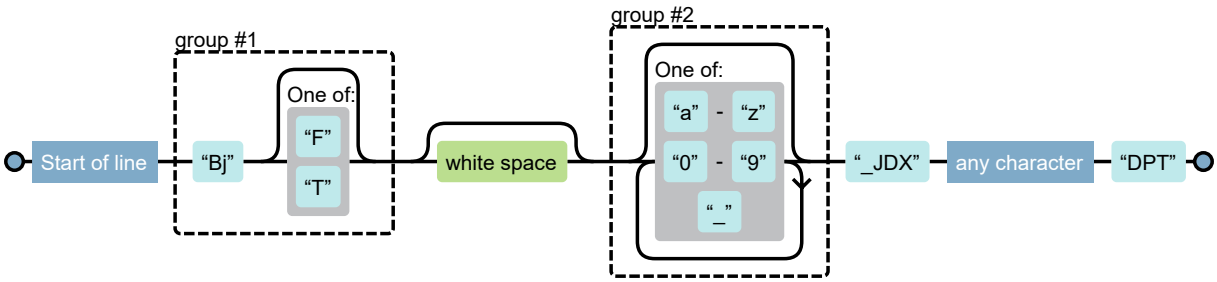

Figure 4.11: Regex to bring DPT file names in a uniform format

The uniform file names were then matched by the main Regex, which extracted *measurement device* and *sample number* and accepted multiple types of *measurement numbers* and *file endings*, which were not relevant for finding the referenced sample. In most cases, the matched sample number could be found directly in the database. If not found, a few further exceptions were checked, for example when the first part of the *measurement number* until the next underscore also belonged to the *sample number*, or when only the

base sample was entered in the metadata table and the specific subsample entry first had to be created.

```
/(.*?)_(.*?)_(\d+|[a-zA-Z0-9]+[_.]\d+)(_JDX)?[.]{1,2}(DPT|csv|CSV|JDX)/
```

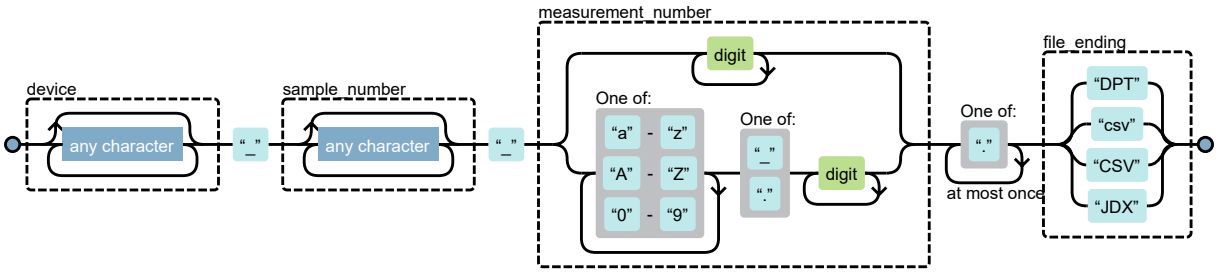

Figure 4.12: Regex for DPT file names

After all exceptions had been checked and a matching sample had been found, the file was loaded and converted from rows of `label,value` into a two dimensional `[[label,value]]` array. Whereas normally all `values` were between zero and one, some samples were also scaled from zero to 100. To find these samples, every spectrum was checked for values higher than ten, as due to measurement errors, some samples scaled to one contained values slightly above one, and scaled down, if needed. Then the spectrum was fully preprocessed and uploaded with `device`, `filename` and `sample_reference`. This concluded the import process. Afterwards all collected errors were saved for later review to manually upload failed samples.

4.6 UI

The UI for the database is provided by an Angular app hosted on the BIC. For the Angular app, the Bosch styled UI components were used, which provided ready-made elements like buttons, inputs or the standard Bosch header. During the project, also a few contributions to this library were made, improving the data input component to have a label and drag and drop functionality, as well as small bug fixes, discovered during development.

The header provides navigation between the different pages, each page being an own Angular component. Figure 4.13 shows the view the users see when visiting the page for the first time. Besides the login page only the documentation tab is visible. As can be seen in the following screenshots, the number of navigation links in the header increases after login depending on the user level. The page saves the credentials in the local storage, so the user is automatically logged in, when visiting the page again.

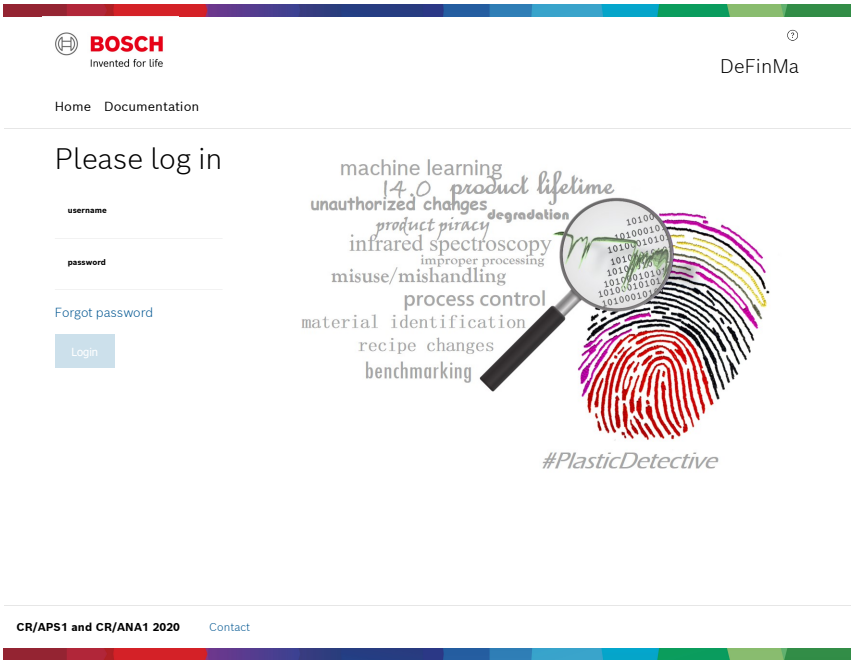

Figure 4.13: Home view before login

4.6.1 Services

The App has a number of services, which provide functions used in multiple components.

`api` is a wrapper for the `HttpClient`, provided by Angular, to make API calls. It provides methods for GET, POST, PUT and DELETE requests to the API routes, which additionally accept the request body data and a callback. Each route adds the corresponding host name to the route, depending on whether the app is in development or production and also the Basic Authorization credentials. Depending on the number of callback function arguments, the result is handled differently. If one argument is supplied, the callback function is called with the response data if the call was successful. If an error is returned, a general *Network request failed* modal is shown to the user. When supplied with two arguments, the callback function is called with response data and error, without any error handling by the service itself. An optional third argument is used for returning the response headers.

`autocomplete` is a service to simplify the use of the Bosch styled autocomplete input field. It provides a search function utilizing the *QuickScore* module for searching the list of available completions by the user's input. In contrast to simply matching results containing the user input, the *QuickScore* module is based on the *Quicksilver algorithm*, which ignores

spelling mistakes and instead puts focus on details like capitals in the returned list of completions.

`data` maintains API resources like `materials`, `suppliers` or `groups`. It provides a `load()` function for the components. The specified resource is loaded, stored in the service and provided for all components. A callback is executed when the resource is loaded to notify the component that the resource is available. If possible, the resource is also provided as an object with the `_id` as a key as this is often needed by the components. Storing the resource as both array and object needs more memory but reduces processing load as otherwise the array had to be filtered for every `_id` request. The usage of the data service improves page load times in comparison to loading resources in the component itself, as otherwise these resources would need to be loaded for every component they are used in. Due to the nature of Angular as a single page application, these resources instead can be shared across all subpages after being loaded once.

`login` provides the authorization for the API of the app. When the main `login()` method is executed, it either checks the given credentials using the `/authorized` method, or otherwise extracts the stored credentials from the local storage. If the credentials are correct, the login status is set, along with the level of the user. The service also provides the `canActivate` method for the Angular router to determine whether the router may access this route based on the user's `level` or not. Further a `logout` method is provided, removing the credentials from the local storage.

`validation` provides together with the `validate` directive custom validations for user input. Therefore the `appValidate` attribute has to be included in the user input field. This applies the correct method to the input field like validation for password, minimum value or unique values. As previously explained, the *Joi* validation created for the back end is directly used for these validation methods.

4.6.2 Auxiliary components

Some components are not used as a routing endpoint by itself, but rather implement functions needed in multiple components.

`img-magnifier` provides a rectangular magnifying glass following the mouse cursor when hovering over an image. Often used in some form on the product images of reselling sites, this functionality is needed as the Bosch layout limits the page contents to a maximum width of 1440 pixels. Big, detailed images like the database structure diagram in Figure 4.2 are still quite small even stretched over the full available content width and can be viewed easier using the `img-magnifier`.

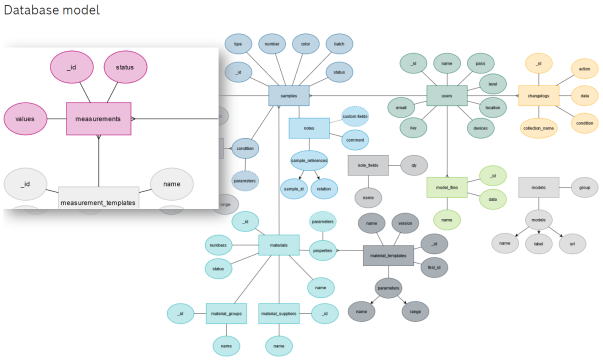

Figure 4.14: Magnifying glass in the database documentation

`rb-array-input` is a component which multiplies the nested input elements, allowing an easy input of array structures with variable length. One nested input field here has to be declared as the controlling field. As soon as the user inputs a value in this field in the last group, the nested inputs are multiplied once more, allowing the user to input another array value. If the contents are deleted again, the last group is removed, in case the last two groups are empty. Empty fields in the middle are not removed but rather filtered out when processing the array, as this behaviour could be confusing for the user. In cases in which the user wants to remove all text to input a new value, the user would otherwise write in the following field after deleting the last character instead of being able to enter a new value.

`rb-icon-button` provides an easier use of the Bosch styled buttons and icons, with consistent spacing between icon and text and a smaller button when no text should follow the icon.

`rb-table` is a wrapper for all nested table elements as there is no dedicated *table* element from the the Bosch styled UI components. Bosch styles are applied and options like text ellipsis and moving the horizontal scroll bar to the top are supported.

4.6.3 Prediction

The *Prediction* page allows developers to quickly give customers a demo of their developed prediction models. A tabbed view shows all `model groups`, and below the model to use for prediction from this group can be selected. A file input allows the user to upload multiple spectrum files and a selection lets the user specify whether the files are remeasurements of the same sample or whether each file is from a different sample. The files are then sent to the corresponding prediction model endpoint and the results are presented for each spectrum, according to the selected case. Figure 4.15 shows the case of remeasurements with additional standard deviation. The *Details* section additionally lists all spectra with their corresponding single predictions.

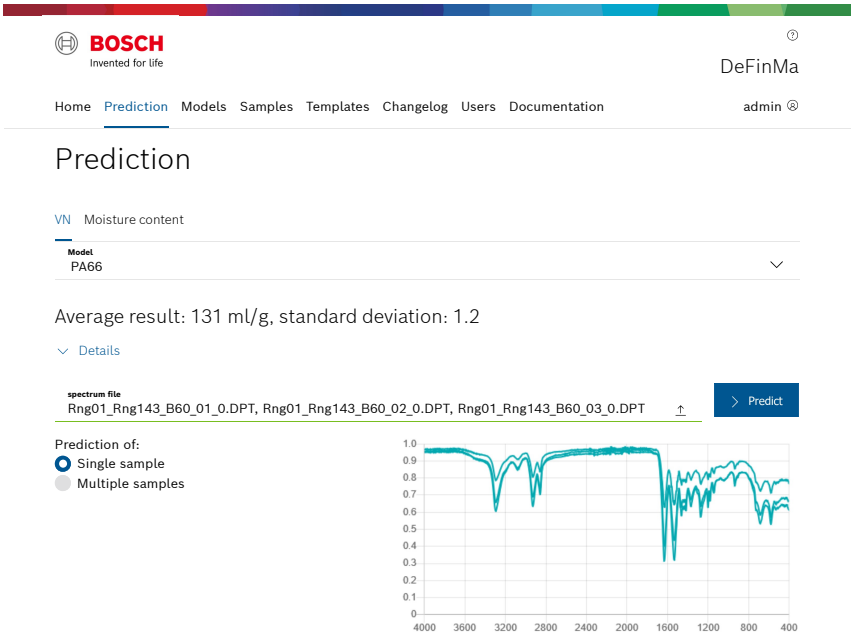

Figure 4.15: *Prediction* page

4.6.4 Models

dev and *admin* users can specify the model entries for the models that should be shown in the *Prediction* dialogue. A list shows all groups and models, which can be edited or deleted. Additionally new entries can be created.

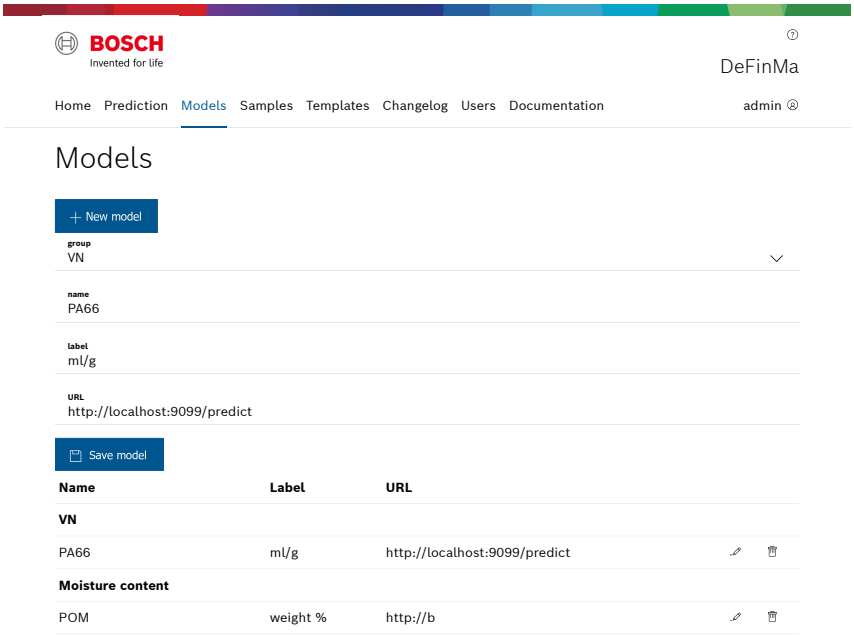

Figure 4.16: *Models* page

4.6.5 Samples

This page is the one used most frequently by data users as well as developers. The table in Figure 4.17 gives an overview of the sample data in the database utilizing the `/samples` route. The data is displayed in pages and the current view (including all pages) can be downloaded by *dev* and *admin* users either in CSV or JSON format using the given back end functionalities. All functions of the `/samples` route can be used with a graphical interface in this component. All set options are included in the query string and automatically sent when the filters are applied to display the data as wished. To ensure fast loading times after changing options, the materials are loaded once, using the `data service`, and displayed from memory instead of requesting the material fields on every request, too.

The sorting option is always visible with a filled arrow next to the column name, which is currently used as a sorting key and unfilled arrows next to all column names which allow sorting, to sort the data by this column either ascending or descending.

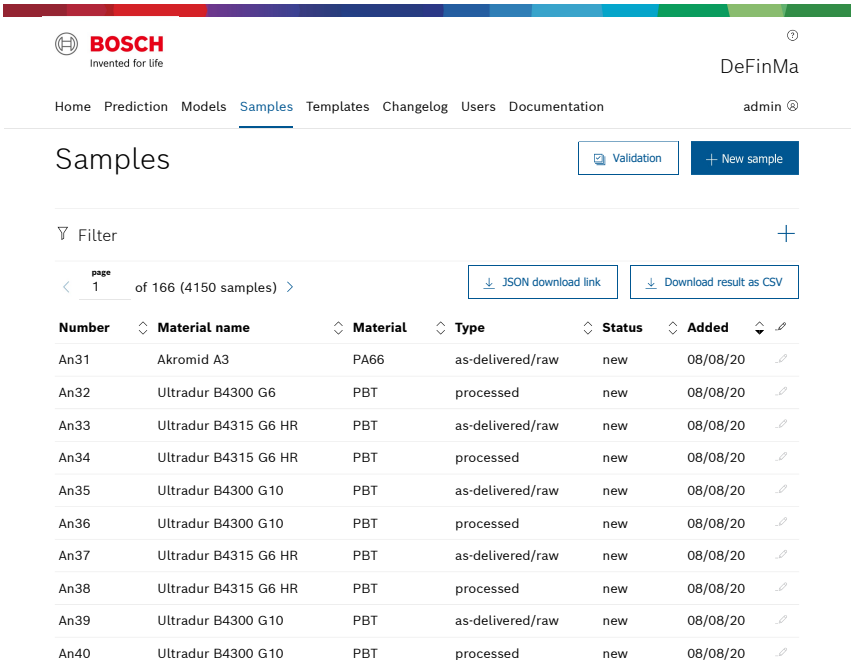

Figure 4.17: Default view of the *Samples* page

All further options are collapsed by default to save space and can be expanded by clicking on the filter icon, shown in Figure 4.18. Here the user can select which status the samples should have. Every user can select the *validated* or *new* status, but only *dev* and *admin* users can also select to show *deleted* samples. Further the number of items per page and the data fields to be shown in the table can be selected. For all selected fields an entry is shown below, to filter the data by this field, when possible. Each filter can be activated manually but is activated, when the user types something in the input field automatically, for more convenience. The comparison mode can be set to the values already explained in Table 4.2 with the drop-down. Most of the filter value input fields are plain input fields. For fields for which the server provides autocomplete data, like `material groups` or `suppliers`, an autocomplete feature is built in. The `type` field is carried out as a drop-down input, as the type can only be one of two values, and for the `added` filter a date input is provided. All options are stored in the local storage, presenting the user with the same view they left off when they last visit the page again.

The *validation* button is only visible for *dev* and *admin* users. Once clicked, every row is preceded by a checkbox. This allows selecting the samples to be validated or selecting all samples by clicking the checkbox in the table header. The selection then has to be confirmed to validate the selected samples or declined to discard the current selection. The same checkboxes also appear when multiple samples are selected for editing, which is initiated by the edit icon in the table header.

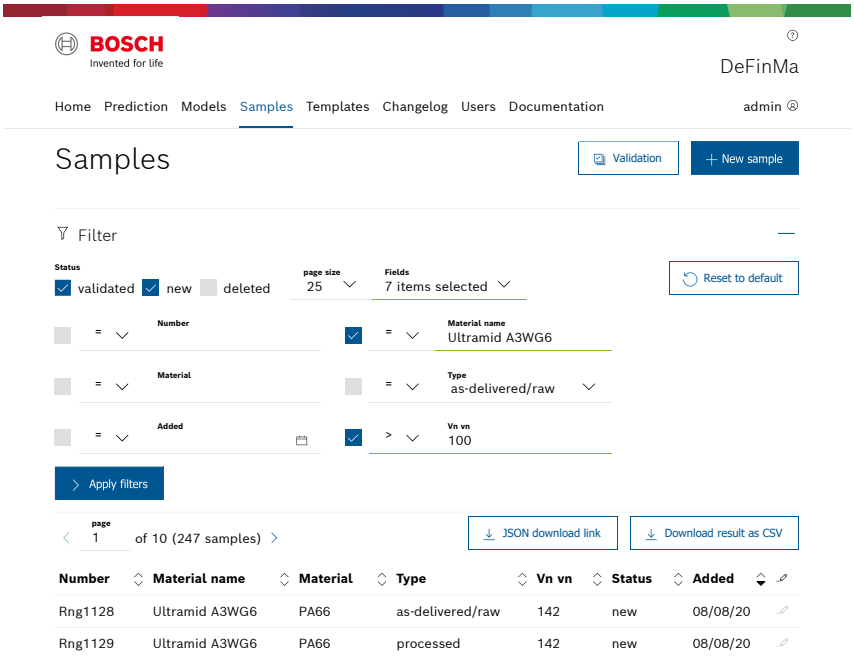

Figure 4.18: Filters on the *Samples* page

4.6.6 Sample

This component is used for either adding new samples or editing existing samples. It can be reached by either clicking the *New sample* button or the edit button next to a sample route from the *samples* component. The *sample* component consists of two parts. When adding a new sample, the first part is the only part displayed at first as can be seen in Figure 4.19. The `material name` field provides autocompletion from the existing entries and automatically expands the *New material* dialogue when an unknown name is entered. Alternatively a button is provided to open the material input fields. All values are instantly validated using the `validation` service. The material `type` lets the user select the corresponding template, changing the fields below to specify the material specific properties.

The sample references are also autocompleted, but in contrast to the other autocomplete fields these results are loaded from the API, instead of from memory, as the number of completions would be too large. As the reference must be a valid sample, the user has to select an entry from the list of suggestions. The relation description, however, is not restricted in any way. The *Additional properties* `key` also provides autocompletion to avoid multiple keys with the same meaning. Here new values are accepted and a new key is created in the database automatically.

After all values are entered and valid, the *Generate sample* button gets clickable. The number input lets the user specify how many samples with the entered base data should be generated. This feature is used frequently, as a measurement series includes multiple samples with the same base properties.

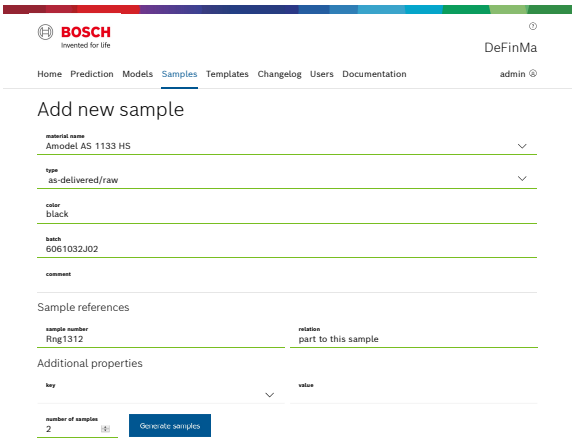

Figure 4.19: Part one of the *New sample* page

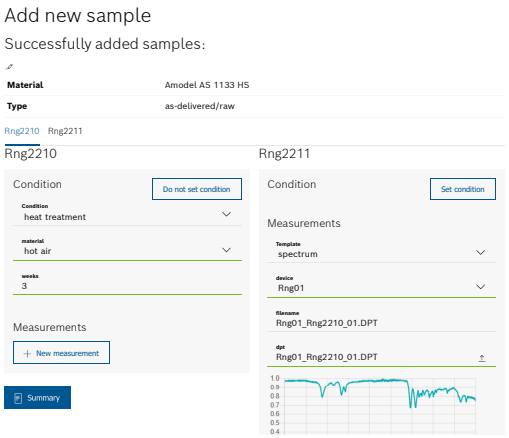

Figure 4.20: Part two of the *New sample* page (summary shortened for display)

The second part of the *sample* component, shown in Figure 4.20, gives a summary of the generated sample base on the top. This part is the default view when editing a sample. Here the first part is accessible by an edit icon next to the summary. Below the summary, all generated sample numbers are listed in tabs together with *condition* and *measurements* of all samples. As these differ from sample to sample in a measurement series, they have to be entered individually.

The *condition* can be selected from one of the available `condition_templates`. For *measurements*, multiple entries can be added. Again a `measurement_template` has to be selected for each entry, and the corresponding fields filled. For the *spectrum* template, the input is optimized as this is the most frequently used measurement. Here the spectrum files can be selected, automatically expanding each file into its own *measurement* and automatically filling the *file name*. Also the required *measurement device* is not validated by the available `template range`, but can only be one of the measurement devices specified in the user profile. After entering all data, clicking on the *Summary* button shows a summary page of all properties entered and a button to save all sample data.

4.6.7 Templates

The *Templates* component is only available for *dev* and *admin* users for managing the `material_`, `condition_` and `measurement_templates`. Next to a selection to switch between the three groups, it features a list of all templates currently available and a button to add a new template at the bottom of the list. The template list item expands on click, revealing all former versions of this template, including the respective properties. The latest version of the template can also be edited to add or remove parameters and edit their range in the same way as shown in Listing 4.3, using the JSON syntax.

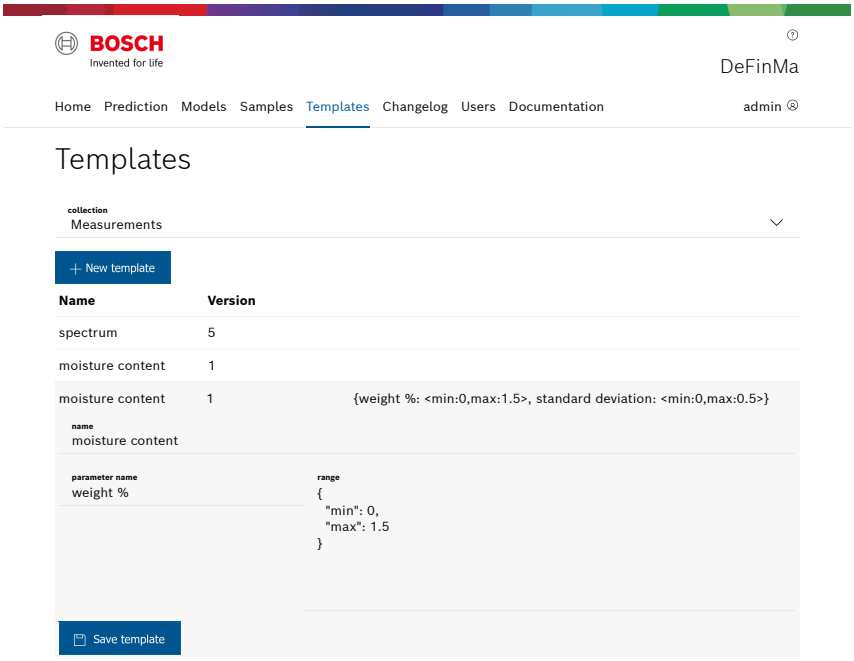

Figure 4.21: *Templates* page

4.6.8 Changelog

This page provides a user-friendly presentation of the changelog data for *dev* and *admin* users, as explained in subsection 4.4.8. A date picker can be used to set the timestamp to the point in time from which the changelog data should start. The page size can also be adjusted.

Using the *next page* button, the user can go back in time of the changelogs, while the date picker automatically updates to the date of the first sample on the page when going to the next page. This avoids skipping a large number of documents, which is not possible in an efficient way in MongoDB, as already explained in subsection 4.4.1. When clicking on a sample, the full log details are presented in a pop-up dialogue.

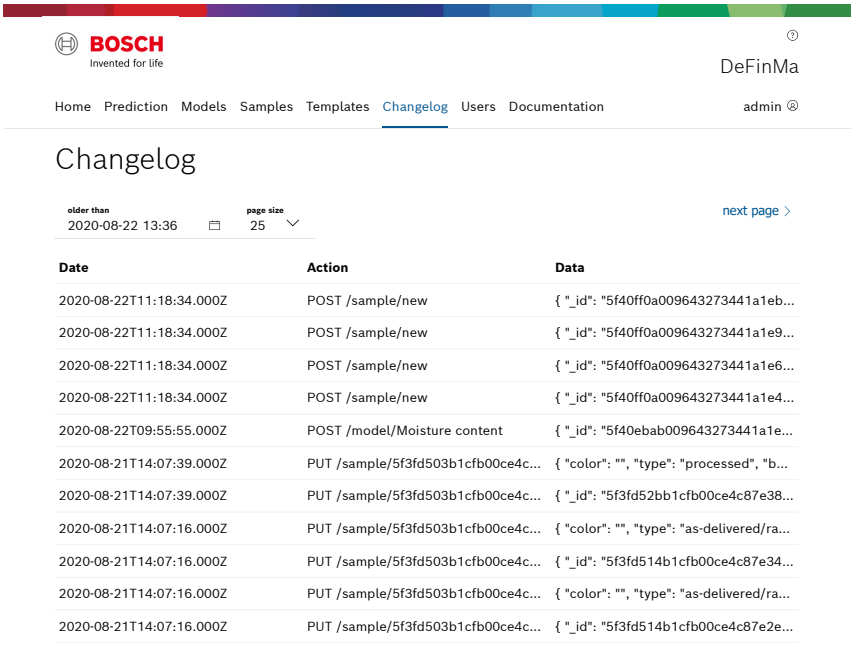

Figure 4.22: *Changelog* page

4.6.9 Users

The *Users* component is accessible only for *admin* users. It shows a list of all existing users including their details, except the password. The details can be changed and a new user account can be created.

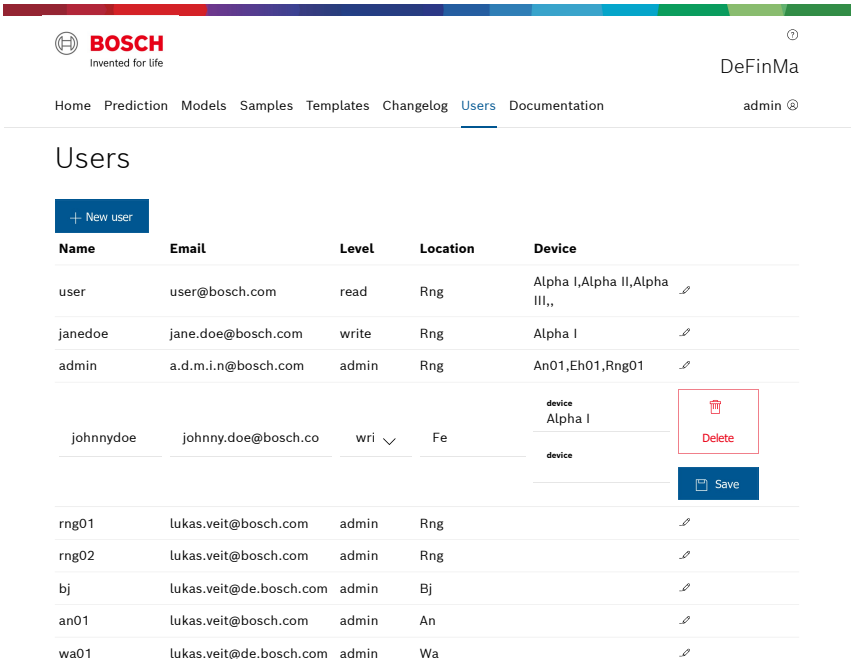

Figure 4.23: *Users* page

4.6.10 Settings

The users can change their account settings themselves by invoking the *Settings* page from the top right avatar icon. This includes changing the *user name*, if the requested alternative is available, *email*, *location* and *devices*, which are assigned to the user. Additionally, the password can be changed.

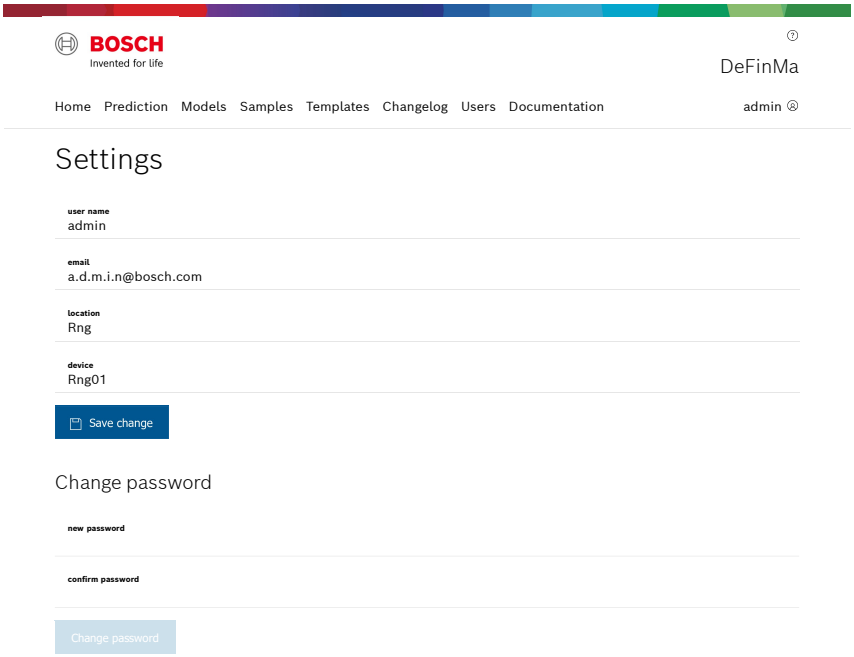

Figure 4.24: *Settings* page

4.6.11 Documentation

The *Documentation* page provides further information about this project. The *General* tab provides a link to the introductory presentation for new users of the UI as well as an overview of the permissions for different user `levels`. Additionally the main links to the API documentation generated from the OAS as shown in section 2.4 as well as the source code repositories are provided. The *Database* tab explains the database structure, showing the diagram from Figure 4.2 as well as a table, explaining all database fields.

5 Conclusion

The task of this project could be completely fulfilled within the given timeframe. A database model specific for the requirement of this project was developed and implemented in a MongoDB on the BIC. All data is accessible via the implemented and documented API routes. All input to the database via the API is validated, guaranteeing data consistency. Access to the data was secured and possible attack vectors mitigated, preparing the application to be hosted publicly.

The UI provides all functions required for the given user groups. The data overview provides everything needed for data maintenance as well as the filter and search options required by developers to generate the exact dataset needed. The generation of a link for the selected dataset makes loading the data for machine learning purposes very convenient. Data upload by other departments is also possible, while the user management ensures that these users only possess the permissions needed. Furthermore all changes are tracked, allowing corrections of mistakes. Deployment of working machine learning models, to provide the customer a first demonstration, is also possible.

During the test phase with other developers from the project the concept of the application proved to be workable. Apart from minor bugs and further feature requests, all of which could be implemented, the application fulfilled the users' needs.

5.1 Prospect

This project was very focused on the data structure and the direct handling of the data. For machine learning models, the actual goal of the DeFinMa project, only a considerably simple interface was provided. In the future, the current architecture can be focused more on utilizing machine learning. One possible idea would be to deploy self-improving models. While the models are currently trained by a developer and the finished model is uploaded, in the future, the training script itself could be hosted in the BIC and regularly train itself, constantly improving along with the steadily increasing data set. Data upload could

also be simplified by machine learning model application. Models for classification and measurement prediction could for example be used to prefill the material of the sample. On the other hand user errors like uploading one wrong spectrum could be detected as the model would classify this sample as another material or predict a moisture content far different from the prediction for all other samples.

While the database structure provides dynamic data input by using templates, the whole application is still pretty focused on the current use case. Bigger changes in the project, and as a consequence thereof the data structure, would require further development in both API and UI. The use of widely popular technologies like MongoDB, Node.js, Express and Angular, however, make extending the current application by another developer straightforward, as the modularity and structure of the codebase is given by these technologies.

It also has to be noted that this is neither the only application inside Bosch collecting data for machine learning nor the only research department having the need to store structured measurement data. In our age of machine learning and big data, a unified solution of course seems to be the best solution in the long term. This project merely presents the first step in data organization and indeed a huge step compared to the previous data management. The next step to migrate data into an other database environment should be feasible as the data now is already structured and well-organized. This project thereby provided a quick solution making current daily tasks in the DeFinMa project easier while preparing the project for the future.

Bibliography

Literature

[1] Paolo Atzeni et al. “Data modeling in the NoSQL world”. In: *Computer Standards & Interfaces* 67 (2020), p. 103149. ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2016.10.003. URL: http://www.sciencedirect.com/science/article/pii/S0920548916301180.

[2] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. “Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1365–1375. ISBN: 9781450341394. DOI: 10.1145/2976749.2978338. URL: https://doi.org/10.1145/2976749.2978338.

[3] I. Dolnák and J. Litvik. “Introduction to HTTP security headers and implementation of HTTP strict transport security (HSTS) header for HTTPS enforcing”. In: *2017 15th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2017, pp. 1–4.

[4] B. Hou et al. “MongoDB NoSQL Injection Analysis and Detection”. In: *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*. 2016, pp. 75–78.

[5] Isatou Hydara et al. “Current state of research on cross-site scripting (XSS) – A systematic literature review”. In: *Information and Software Technology* 58 (2015), pp. 170–186. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.07.010. URL: http://www.sciencedirect.com/science/article/pii/S0950584914001700.

[6] Priya Jyotiyana and Saurabh Maheshwari. “Techniques to Detect Clickjacking Vulnerability in Web Pages”. In: *Optical and Wireless Technologies*. Ed. by Vijay Janyani et al. Singapore: Springer Singapore, 2018, pp. 615–624. ISBN: 978-981-10-7395-3.

[7] Engin Kirda et al. "Client-side cross-site scripting protection". In: *Computers & Security* 28.7 (2009), pp. 592–604. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2009.04.008. URL: http://www.sciencedirect.com/science/article/pii/S0167404809000479.

[8] Dominic Lingenfelser and Elisabeth Lotter. *Digital Fingerprint of plastics*. Dec. 2019.

[9] Haya Shulman. "Pretty Bad Privacy: Pitfalls of DNS Encryption". In: *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. WPES '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 191–200. ISBN: 9781450331487. DOI: 10.1145/2665943.2665959. URL: https://doi.org/10.1145/2665943.2665959.

[10] Lukas Veit. *Evaluation and improvement of an existing database for sensory and analytical data*. June 2020.

Webpages

[11] *About Node.js*. URL: https://nodejs.org/en/about/ (visited on 08/01/2020).

[12] *Aggregation*. 2020. URL: https://docs.mongodb.com/manual/aggregation/ (visited on 08/09/2020).

[13] *Angular*. URL: https://angular.io/ (visited on 08/01/2020).

[14] *API*. 2020. URL: http://bsonspec.org/ (visited on 08/02/2020).

[15] *Best Practices in API Design*. 2016. URL: https://swagger.io/blog/api-design/api-design-best-practices/ (visited on 08/22/2020).

[16] *BSON*. 2020. URL: http://bsonspec.org/ (visited on 08/02/2020).

[17] *Content Security Policy Level 2*. 2016. URL: https://www.w3.org/TR/CSP2/ (visited on 07/25/2020).

[18] *CSS Introduction*. URL: https://www.w3schools.com/Css/css_intro.asp (visited on 08/01/2020).

[19] *Docs*. 2020. URL: https://helmetjs.github.io/docs/ (visited on 07/19/2020).

[20] *Documentation*. URL: https://www.typescriptlang.org/docs/home.html (visited on 08/01/2020).

[21] *Hacking NodeJS and MongoDB*. 2014. URL: https://blog.websecurify.com/2014/08/hacking-nodejs-and-mongodb.html (visited on 08/02/2020).

[22] *HTML Introduction*. URL: https://www.w3schools.com/html/html_intro.asp (visited on 08/01/2020).

[23] *Introducing JSON*. URL: https://www.json.org (visited on 08/01/2020).

[24] *Introduction to MongoDB*. 2020. URL: https://docs.mongodb.com/manual/ (visited on 08/02/2020).

[25] *Introduction to the Angular Docs*. 2020. URL: https://angular.io/docs (visited on 07/05/2020).

[26] Alex Klaus. *Preventing Cross-site scripting (XSS) attacks in Angular and React*. 2019. URL: https://alex-klaus.com/protecting-angular-from-xss-attacks-with-csp/ (visited on 07/05/2020).

[27] *MongoDB Tutorial*. 2020. URL: https://hapi.dev/module/joi/api (visited on 08/03/2020).

[28] *OpenAPI Specification*. 2018. URL: https://swagger.io/specification/ (visited on 08/02/2020).

[29] *Referrer Policy*. 2017. URL: https://www.w3.org/TR/referrer-policy// (visited on 08/01/2020).

[30] *Regular expressions*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions (visited on 08/22/2020).

[31] *Security*. 2020. URL: https://angular.io/guide/security (visited on 07/05/2020).

[32] *Testing*. 2020. URL: https://angular.io/guide/testing (visited on 08/02/2020).

[33] *The different types of software testing*. 2020. URL: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (visited on 08/02/2020).

[34] Patrick Vananti. *Bypassing Content-Security-Policy with DNS prefetching*. 2016. URL: https://blog.compass-security.com/2016/10/bypassing-content-security-policy-with-dns-prefetching/ (visited on 07/28/2020).

[35] *X-XSS-Protection*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection (visited on 08/01/2020).