HTBLA Grieskirchen
4710 Grieskirchen, Parzer Schulstraße 1

Fachrichtung Informatik

Schuljahr 2021/2022

DIPLOMARBEIT

Gesamtprojekt

Debugging Interface für Wasserstoffsysteme

Ausgeführt von:

Vasilije Maglov, 5AHIF-13

Fabian Psutka, 5AHIF-15

Julian Schwaiger, 5AHIF-18

Betreuer/Betreuerin:

Dipl.-Inf. Torsten Welsch

Grieskirchen, am 30.03.2022

Abgabevermerk:
Datum: 30.03.2022

Betreuer/Betreuerin: Dipl.-Inf Torsten Welsch

Erklärung gemäß Prüfungsordnung

„Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle den benutzten Quellen wörtlich entnommenen Stellen als solche kenntlich gemacht habe."

Grieskirchen, 30.03.2022

Verfasser/Verfasserinnen:

Vasilije Maglov

Fabian Psutka

Julian Schwaiger

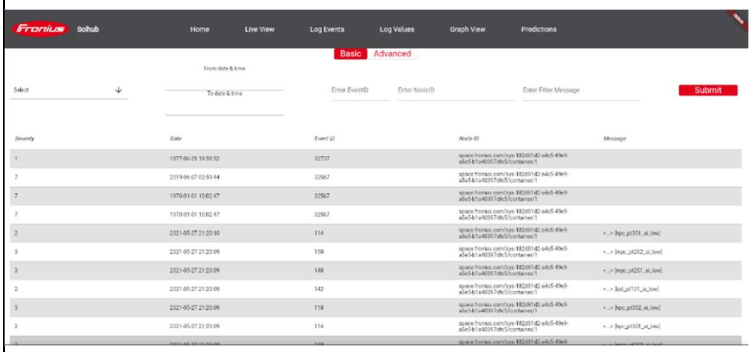
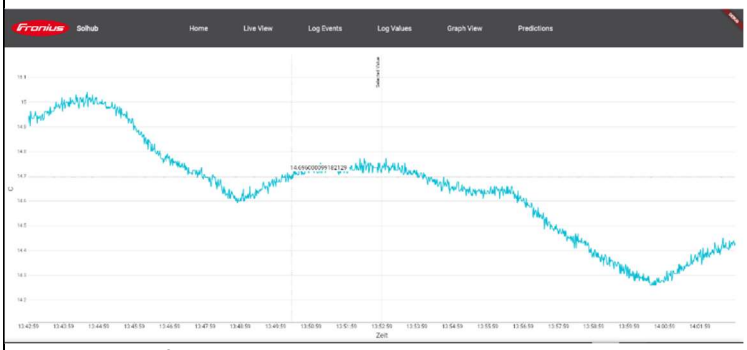
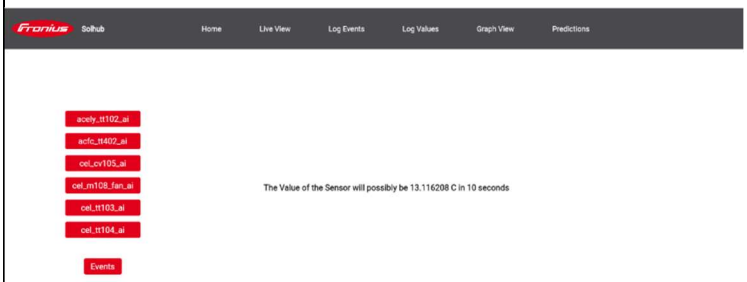
DIPLOMARBEIT
DOKUMENTATION

Namen der Verfasser	Vasilije Maglov Fabian Pstuka Julian Schwaiger
Jahrgang **Schuljahr**	5AHIF 2021/2022
Thema der Diplomarbeit	Debugging Interface für Wasserstoffsysteme
Kooperationspartner	Fronius International GmbH

Aufgabenstellung	Es soll eine Web-App programmiert werden, welche die Auswertung und Überwachung der Daten des sogenannten Solhub-Systems ermöglicht. Hierfür soll eine eigene Filter-Query definiert werden, um das Filtern der Daten zu vereinfachen. Darüber hinaus soll mit Hilfe von Predictive Maintenance die Wartung der Anlage durch die Möglichkeit, Vorhersagen zu treffen, vereinfacht werden.

Realisierung	Das Frontend wurde mit dem Framework Flutter umgesetzt, welches auf der Sprache Dart basiert. Für das Backend wurde ein Node.js Server verwendet. Die Filter-Query wurde, wie das Backend, in JavaScript implementiert. Die Kommunikation zwischen dem Frontend und dem Backend geschieht über REST-Schnittstellen. Der Teil der Predictive Maintenance wurde mit Hilfe von Keras und Tensorflow in Python umgesetzt.

Ergebnisse	Als Ergebnis der Diplomarbeit haben wir eine Website erstellt, ein Backend mit einer selbst definierten Filter-Query-Language und System zur Predictive Maintenance. Die Website verwendet das Backend, um die Daten des Solhub-Systems leicht verständlich anzuzeigen. Auch können Graphen aus diesen Daten erstellt werden und die Daten mit der Filter-Query leicht gefiltert werden. Darüber hinaus können mit Predictive Maintenance Vorhersagen über zukünftige Messwerte und Events gemacht werden.

Typische Grafik, Foto, etc. (mit Erläuterung)	 Hier wird die Tabellenansicht der Event-Daten dargestellt. Man sieht oben die verschiedenen Filteroptionen, die die Datensätze darunter einschränken können.  Die zweite Grafik zeigt einen Graphen, welcher erstellt wird, wenn der User einen Datensatz auswählt. Der Chart zeigt die Messwerte eines Sensors über einen bestimmten Zeitraum.  Hier sieht man schlussendlich die Predictions-Seite, auf welcher die Vorhersagen für verschiedenste Sensoren und Events getroffen werden.

Teilnahme an Wettbewerben, Auszeichnungen	Teilname am „Young Science Inspiration Award“ https://youngscience.at/de/awards-und-guetesiegel/young-science-inspiration-award-1

Möglichkeiten der Einsicht-nahme in die Arbeit	In der Schulbibliothek

Approbation (Datum/Unterschrift)	Prüfer/Prüferin	Direktor/Direktorin

DIPLOMA THESIS
DOCUMENTATION

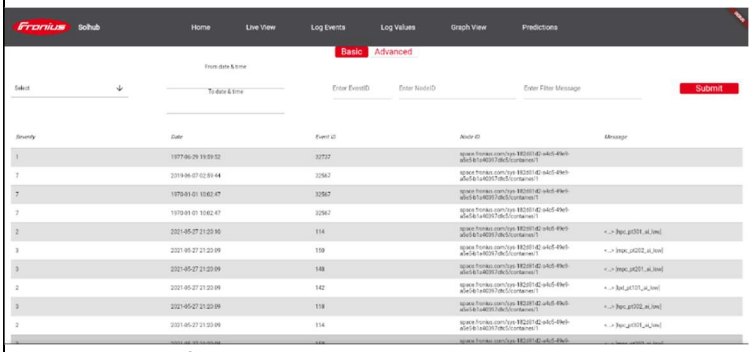
Author(s)	Vasilije Maglov Fabian Pstuka Julian Schwaiger
Form **Academic year**	5AHIF 2021/2022
Topic	Debugging Interface für Wasserstoffsysteme
Co-operation partners	Fronius International GmbH

Assignment of tasks	A web app is to be programmed that enables the evaluation and monitoring of the data of the so-called Solhub system. For this purpose, a filter query is to be defined in order to make the filtering of the data easier. In addition, predictive maintenance is to be used to simplify the maintenance of the system through the possibility of making predictions.

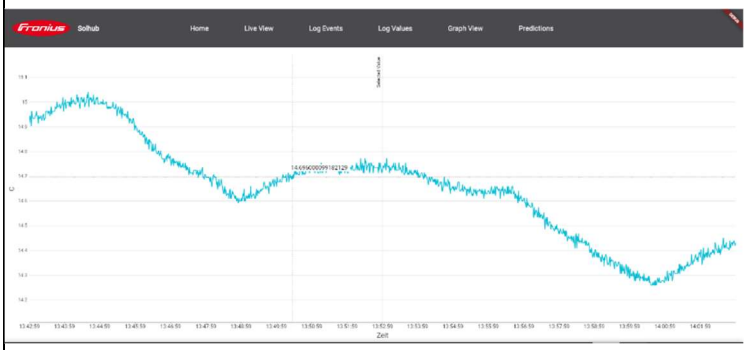
Realisation	The frontend was implemented with the Flutter framework, which is based on Dart. A Node.js server was used for the backend. The filter query, like the backend, was implemented in JavaScript. The communication between the frontend and the backend is done via REST. The predictive maintenance part was implemented in Python with the help of Keras and Tensorflow.

Results	As a result of the diploma thesis, we created a website, a backend with a self-defined filter query language and a predictive maintenance system. The website uses the backend to display the data of the Solhub system in an easily understandable way. The data can also be visualized with graphs and easily filtered with the filter query. In addition, Predictive Maintenance can be used to make predictions about future values and events.

Illustrative graph, photo, etc.
(incl. explanation)



The table view of the event data is shown here. At the top you can see the different filter options that can limit the records below.



The second chart shows a graph that is created when the user selects a data set. The chart shows the measured values of a sensor over a certain period of time.

Here you can see the Predictions page, where the predictions for various sensors and events can be made.

Participation in competitions Awards	Participation in the „Young Science Inspiration Award“ https://youngscience.at/de/awards-und-guetesiegel/young-science-inspiration-award-1

Möglichkeiten der Einsicht-nahme in die Arbeit	In the school library

Approval (date/signature)	Examiner	Head of College/Department

Inhaltsverzeichnis

1 Einleitung

Für jede neue Technologie muss es einen Weg geben diese zu überprüfen, Fehler zu finden und zu beheben. Jedoch ist dies meist schwierig mit größeren Maschinen oder Themengebiete, für die es keine bzw. wenig Messinformationen gibt. Die Aufgabe dieser Diplomarbeit war es eine Webseite zu bauen, die es für den User erleichtern soll, die Daten des Wasserstoffsystems (Solhub von Fronius) einzusehen, darin Fehler zu finden und diese zu beheben. Die Daten werden von hunderten von Sensoren erfasst und in die systeminterne Datenbank gespeichert. Für den User besteht dann die Möglichkeit, die Daten über unsere Software mit einer leicht verständlichen, selbst definierten Sprache zu filtern und die Ergebnisse in einer Graphenansicht anzeigen zu lassen. Neben diesen Funktionen kann der User auch verschiedene Vorhersagen mit Hilfe von Predictive Maintenance treffen lassen, welches in unserer Seite mit eingebaut ist.

Das System, mit welchem dieses Projekt am Ende eine Symbiose eingehen soll, ist Solhub von Fronius. Hierbei handelt es sich um ein System, um aus Sonnenenergie Wasserstoff herzustellen. Da so produzierter Wasserstoff auf lange Zeit gesehen von Experten [1] als eine gute und grüne Energiequelle geseshen wird, jedoch die CO_2-neutrale Produktion oft ein Problem darstellt, forscht Fronius seit Jahren an neuen Wegen diese Herausforderungen zu lösen. Die Lösung ist ihr Solhub System. Es arbeitet mit Solarstrom, um aus Wasser Wasserstoff herzustellen, welcher dann beispielsweise für Wasserstofffahrzeuge verwendet werden kann [2].

2 Auftraggeber

Die Diplomarbeit wurde zusammen mit der Firma Fronius ausgearbeitet. Fronius‘ Hauptgebiete sind die Forschung, Entwicklung und Produktion neuer Technologien in den Bereichen Solar Energy, Perfect Charging und Perfect Welding und vielen mehr, welche weltweit verwendet werden. Ein Teilbereich ist dabei die Wasserstoffmobilität. Unsere Kontaktpersonen waren Christian Kasberger und Joachim Danmayr, mit welchen wir die Anforderungen des Projektes ausgebarbeitet und definiert haben.

3 Architektur

Unsere Software wird grundlegend in drei Teile aufgeteilt: Frontend, Backend und Predictive Maintenance. Ein Ziel von Fronius ist es, dass die einzelnen Teile der Software unabhängig voneinander laufen, angepasst und anderweitig eingesetzt werden können. Die Kommunikation zwischen den einzelnen Modulen findet hierbei zwischen Front- und Backend über REST (Siehe

Punkt 6.3) statt. Das Backend kommuniziert mit dem Predictive-Maintenance-Teil, indem ein *Python*-Programm Vorhersagen trifft und diese an das Backend zurück liefert.

3.1 Backend

Das Backend wird über eine REST-API angesteuert. Als Sprache verwenden wir *JavaScript*, was auf einem *NodeJS*-Server läuft und uns mit dem *Node-Module ExpressJS* die Tools für eine REST-API bietet. Die Requests geben hierbei entweder Values oder Events zurück, um die Anzeige dieser Werte zu ermöglichen. Über Query-Parameter können Filter mitgegeben werden, um die Suchergebnisse einzugrenzen.

3.2 Frontend

Das Frontend benutzt Flutter als Sprache, welche auf Dart basiert. Dieses baut eine Verbindung mit dem Backend auf, wodurch wir dann unsere Daten erhalten. Die Daten werden dann auf den jeweiligen Seiten verarbeitet und angezeigt.

3.3 Predictive Maintenance

Predictive Maintenance ist die Praxis des Vorhersagens von Wartungsarbeiten. Damit kann die Dauer gemeint sein, die ein Teil noch durchhält, bis es gewartet werden muss oder eine Art von Kennzahl, um zu erkennen, wie weit der Verschleiß schon vorangeschritten ist.

Vonseiten Fronius gab es zum Predictive Maintenance keine Vorgaben, was also viel Spielraum zuließ. Wir entschieden uns dazu die Vorhersage von verschiedenen Werten der verschiedenen Sensoren zu ermöglichen. Des Weiteren entschieden wir uns dazu, alle zur Vorhersage nötigen *Models* (die später noch genauer erläutert werden) in einen simplen Ordner, sowie die Logik zum Tätigen einer Vorhersage in einer simplen *Python*-Datei auszulagern. Dazu soll sich an den Technologien *Tensorflow* bzw. *Keras* bedient werden und wie viele andere *Machine Learning* Projekte natürlich alles in *Python* geschrieben werden.

4 Individuelle Aufgabenstellungen

4.1 Julian Schwaiger

Seine Aufgabe war es, zuerst einmal ein Backend für den Zugriff auf die Daten des Solhub-Systems, welche in einer *MongoDB*-Datenbank gespeichert werden, für das Frontend zu programmieren. Dieses Backend sollte dabei in *NodeJS* umgesetzt werden. Des Weiteren sollte eine Query-Language für das Filtern der Daten entworfen und umgesetzt werden. Diese wurde in *JavaScript* realisiert, um die Verwendung auf Backend-Ebene zu erleichtern. Sie bietet die Möglichkeit, syntaktisch richtige Texte in *MongoDB*-Queries für Values und Events (Siehe Punkt 7) umzuwandeln.

4.2 Vasilije Maglov

Seine Aufgabe bestand darin, eine Frontend-Seite zu bauen, um die Daten auslesen zu können, die über das Backend bereitgestellt werden. Dieses Frontend besteht aus mehreren Teilen: Einer Seite, die via Tabellenansicht die Werte der einzelnen Bauteile im Solhub ausgibt, einer weiteren Seite, die via Tabellenansicht alle Events des Solhub-Systems anzeigt und einer letzten Seite, die für die Ausführung der Predictive-Maintenance-Methoden verantwortlich ist. In den jeweiligen Tabellenansichten sollte es möglich sein, einen der dort verfügbaren Werte auszuwählen und dafür einen Graphen zu erstellen, der die dazugehörigen historischen Daten anzeigt.

4.3 Fabian Psutka

Seine Aufgabe bestand darin, ein System zur Predictive Maintenance zu entwickeln. Predictive Maintenance umfasste in diesem Fall ein Deep-Learning-System, das, entwickelt und trainiert mit Hilfe von *Tensorflow* bzw. *Keras*, dazu in der Lage ist, verschiedene Zeitspannen und Werte vorherzusagen. Es sollte als eine Art Pionierprojekt für Fronius dienen, weshalb auch keine konkreten Vorgaben zu Technologien gemacht wurden.

5 Projektmanagement

Um ein Projekt gut umzusetzen, muss der Ablauf realistisch geplant werden. Deshalb ist das Projektmanagement ein wichtiger Teil unserer Diplomarbeit. Die Planung für dieses Projekt fing schon früh an, genauer gesagt im Juli 2020, wobei zu dieser Zeit beim Auftraggeber Fronius angefragt wurde, ob es überhaupt möglich ist, eine Kooperation im Zuge einer Diplomarbeit bei

ihnen zu einzugehen. In den Folgemonaten kam es zu Besprechungen und Telefonaten mit unseren Ansprechpartnern Kasberger Christian und Danmayr Joachim bezüglich der Aufgabenstellung und die Umsetzung im Zuge eines vierwöchigen Praktikums bei Fronius. Am 12. Oktober 2020 wurde ein Anforderungsdokument vom Auftraggeber erstellt, um die Themenstellung und den Umfang zu definieren. Damit letzterer für drei Personen gerechtfertigt war, fügten wir in Absprache mit Fronius noch den Predictive-Maintenance-Teil zur ursprünglichen Aufgabenstellung hinzu.

Bevor das Praktikum begann, wurden im Vorhinein Meilensteine für die Diplomarbeit definiert und erstellt, um einen Überblick über die zu erreichenden Ziele zu gewinnen. Diese werden im folgenden Abschnitt präsentiert und nähert erklärt:

5.1 Meilensteine

Um einen Überblick über die Ziele und den Fortschritt des Projektes zu erhalten, haben wir folgende Meilensteine definiert.

Tabelle 1: Meilensteine

#	Meilenstein	Fertigstellung Soll	Fertigstellung Ist
1	Backend mit *NodeJS* umsetzen (Schwaiger)	27.09.2021	27.07.2021
2	In Flutter Libraries einlesen (Maglov)	27.09.2021	08.07.2021
3	Mathematik und Funktionsweise hinter Tensorflow verstehen (Psutka)	27.09.2021	08.07.2021
4	Datenaufbereitung (Psutka),	25.10.2021	19.07.2021
5	Query Language Schemen definiert (Schwaiger)	25.10.2021	16.07.2021
6	Frontend mit Backend verbunden (Maglov)	25.10.2021	22.07.2021
7	Query Language Filter umsetzen (Schwaiger)	22.11.2021	23.07.2021

8	Graphen im Frontend implementiert (Maglov)	22.11.2021	27.07.2021
9	KI umgesetzt und trainiert (Pstuka)	22.11.2021	27.07.2021
10	Qualitätssicherung Backend (Schwaiger)	10.01.2022	30.07.2021
11	Restliches Frontend mit Flutter umsetzen (Maglov)	10.01.2022	29.07.2021
12	Query Language in Website intergrieren (Maglov)	10.01.2022	21.07.2021
13	Qualitätssicherung der KI (Pstuka)	10.01.2022	30.07.2021
14	Qualitätssicherung der Webseite (Maglov)	14.02.2022	30.07.2021
15	Deployment der Webseite (Maglov)	14.02.2022	30.07.2021
16	Dokumentation der Website (Maglov)	14.02.2022	15.03.2022
17	Dokumentation Query-Language (Schwaiger)	14.02.2022	15.03.2022
18	Dokumentation Predictive Maintenance (Psutka)	14.02.2022	15.03.2022
19	Fertigstellen der Diplomarbeit	14.03.2022	22.03.2022

Die Meilensteine wurden sicherheitshalber über die längere Zeitdauer der gesamten fünften Klasse verteilt, um etwaigen Unwägbarkeiten während des Praktikums begegnen zu können. Durch das Praktikum war es uns jedoch möglich, die meisten Meilensteine innerhalb des Monats Juli abzuarbeiten.

6 Technologien

6.1 Dev Container

Dev Container sind sehr simpel: Es handelt sich bei ihnen um ein Feature von Visual Studio Code, welches es mit Hilfe von *Docker* ermöglicht, alle Dependencies, Libraries etc., die für das Arbeiten an einem Programm benötigt werden, in einem simplen *Docker-Image* zu verpacken und auf einer neuen Maschine laufen zu lassen. Die *Remote Development Extension* von Visual Studio übernimmt dann, solange *Docker* auf dem System installiert ist, das Bauen des Dev Containers und die Herstellung einer Verbindung zu diesem. Alle Terminals in Visual Studio Code sind dann nicht mehr auf der Host-Maschine, sondern in dem Docker-Image [3].

6.2 Docker

Docker ist ein Projekt mit verschiedensten Tools, welche sich darum kümmern, dass das Deployment und die Verteilung von Software durch sogenannte Container erleichtert wird. Dies wird durch Virtualisierung ermöglicht [4].

Ein Container ist eine Einheit für Software, in welcher ihr Code und alle Dependencies verpackt sind. Ein solcher Container kann leicht geteilt werden und mit verschiedenen Tools wie der Docker-Engine ausgeführt werden. Container arbeiten isoliert von ihrem Host-Environment und funktionieren daher unabhängig vom Host-System immer gleich [5].

6.3 REST

REST ist ein Software-Architektur-Style für APIs. Sogenannte *RESTful* APIs können auf verschiedene Arten implementiert werden. Grundlegend geht es bei einer *RESTful* API darum, dass eine Repräsentation von einer Ressource an einen Client geschickt wird. Dieser muss nach dieser Ressource über ein HTTP-Request anfragen. Die Daten können dann in verschiedensten Formaten zurückgeschickt werden. Meistens handelt es sich hier um JSON, HTML, oder Plain Text [6].

6.4 MongoDB

MongoDB ist ein dokumentorientiertes NoSQL-Datenbankmanagementsystem, das in der Programmiersprache C++ geschrieben ist. Die Datenbank selbst ist dokumentorientiert, speichert also JSON-ähnliche Dokumente. Mit dieser Art der Datenspeicherung können Dokumente schemalos gespeichert werden und ermöglichen das Speichern von komplexen verschachtelten Dokumenten, welche trotzdem leicht und schnell abrufbar sind [7].

In unserem Fall wird eine *MongoDB* Datenbank von dem Solhub-System verwendet, um alle Sensor- und Event-Daten (Siehe Punkt 7) zu speichern.

6.5 JavaScript

ECMA Script, oder auch *JavaScript*, ist eine Skriptsprache, welche ursprünglich dafür entwickelt wurde, HTML dynamisch anzupassen. Durch Tools wie *NodeJS* wird *JavaScript* heute jedoch auch außerhalb von Browsern für andere Zwecke wie Webserver und sogar Desktop-Applikationen verwendet.

Wir verwenden *NodeJS*, um *JavaScript* für unser Backend benutzen zu können. Außerdem wird *JavaScript* auch für die die Filter-Query verwendet.

6.5.1 NodeJS

NodeJS ist eine plattformübergreifende Open-Source-JavaScript-Laufzeitumgebung, mit welcher man JavaScript unabhängig von einem Webbrowser ausführen kann. Ein beliebter Anwendungsfall für *NodeJS* ist das Betreiben von Webservern. Die Architektur wurde von Google Chromes *JavaScript Laufzeitumgebung V8* genommen und ermöglicht daher sehr ressourcenfreundlich viele Netzwerkverbindungen gleichzeitig. Mit der Hilfe des *Node-Package-Manager* werden sogenannten *Node-Modules* verwaltet. Diese Node-Modules bestehen aus fertigen Libraries und Tools, wie zum Beispiel Express, welche mit dem Manager verwaltet werden können [8].

Durch NodeJS können wir JavaScript auf einem Server laufen lassen und damit ein leichtes, sehr simples Backend mit den Vorteilen von JavaScript und verschiedenen *Node-Modules* aufbauen.

6.5.2 Express

Eines dieser *Node-Modules* ist *Express*. Hierbei handelt es sich um ein in *JavaScript* umgesetztes Open-Source-Projekt Webframework. Es wird meistens für REST APIs in *JavaScript* verwendet. Für diesen Anwendungszweck bietet *Express* verschiedenste HTTP Methoden und Middleware Funktionen wie einen Router [9].

Express wird bei uns verwendet, um das Backend einfach und simpel in *NodeJS* umzusetzen.

6.6 Flutter

Flutter (auch bekannt unter dem Codenamen „Sky“) ist eine Open-Source UI-Programmiersprache basierend auf Dart. Sie wird benutzt um plattformübergreifende Applikationen von einer

einzigen Codebasis für Android, Linux, Windows, IOS etc. zu erstellen. Flutter wurde 2017 von Google veröffentlicht [10] [11].

6.6.1 Dart

Dart ist eine Programmiersprache, welche für die Client-Entwicklung entworfen wurde. Sie wurde von Google entwickelt und kann für das Bauen von Server- und Desktop-Applikationen verwendet werden. Dart ist eine objektorientierte und klassenbasiert Programmiersprache mit C-Stil-Syntax, welche entweder in nativem Code oder JavaScript kompiliert werden kann [12] [13].

6.7 Python

Python ist eine multiparadigmatische Programmiersprache, die sich besonders im *Machine- und Deep-Learning* großer Beliebtheit erfreut [14]; Es war deshalb eine natürliche Wahl für die Predictive Maintenance, außerdem sind *Keras* und *Tensorflow* nur für *Python* verfügbar.

6.8 Tensorflow

Tensorflow ist ein *Machine-Learning*-Framework, das von Google entwickelt wurde und von *Keras* genutzt wird. *Tensorflow* übernimmt die Teile des *Machine-/Deep-Learnings*, die eher hardwarenah sind, wie zum Beispiel die Option der Berechnungsbeschleunigung mit Hilfe einer GPU, oder auch die Berechnung der Werte der verschiedenen Tensoren in ihren *Layers*, die die Bestandteile eines *Models* sind, was später noch erklärt wird. Diese Tensorberechnungen müsste man ansonsten „händisch“ berechnen, wofür man allerdings Mathematikkenntnisse auf Studienniveau benötigen würde.

6.9 Keras

Keras ist, wie bereits vorher erwähnt, ein weiteres *Machine-/Deep-Learning*-Framework, das von François Chollet entwickelt wurde [15]. Während *Tensorflow* hardwarenäher operiert und Tensorberechnungen durchführt, dient *Keras* vielmehr dem Programmierer als Hilfe. Das fertige Produkt eines Trainingsdurchlaufs und die grundlegende Struktur, die später der Vorhersage dient, ist das *Model*. Dieses *Model* wiederum besteht aus mehreren *Layers*, die die Daten verändern , die dem System als Input zugeführt werden. Anschließend kann man das trainierte *Model* speichern, um es später wieder zu verwenden.

Zu Beginn von Keras war das sogenannte *Backend*, also eine eigene Programm-Bibliothek oder ein System, das die Tensorberechnungen übernimmt, sehr frei wählbar, doch 2017 änderte sich das, als *Keras* von *Google* in Tensorflow bzw. verkehrt herum integriert wurde [16].

7 Daten

Das Solhub-System verfügt aktuell über ca. 200 Sensoren. Diese überwachen das Systems und lösen sogenannte *Events* aus, wenn bestimmte Zustände, wie eine Überhitzung oder der Ausfall von verschiedenen Systemen, eintreten. Diese Daten werden alle zusammen im bestehenden Solhub-System in einer *MongoDB*-Datenbank in zwei getrennten Collections, Values und Events, gespeichert. Ein weiteres Feature, was sich beide Collections zu Nutze machen, sind verschachtelte Dokumente: In beiden Typen von Dokumenten, *Events* und *Values*, sind weitere verschachtelte Dokumente enthalten. Diese vereinfachen das Speichern der Daten, benötigen jedoch bei Abfragen, vor allem wenn nach Werten in diesen verschachtelten Dokumenten gesucht wird, komplexere Queries. Auf diese gehen wir später im Filter-Query-Teil (Siehe Punkt 8.7) genauer ein.

7.1 Events

Ein Teil der Daten, welche für das Auswerten des Solhub-Systems wichtig sind, sind die Events. Sie haben einen Startpunkt und einen Endpunkt, wofür jeweils ein Dokument in der Datenbank erstellt wird. Diese Dokumente haben verschiedene Attribute, wobei manche davon verschachtelt in einem Data-Objekt enthalten sind. Diese Attribute lauten:

Tabelle 2: Attribute Event Datensatz

Attributname	Beschreibung
`timestamp`	Beinhaltet das Datum und die Zeit an dem das Event stattgefunden hat. Das Format ist ein Unix Timestamp, was als Sekunden seit dem 1. Januar 1970 in der GMT Zeitzone (+0:00) dargestellt wird. Beispiel: Der Wert 1617879178 entspricht dem 08. April 10:52:58 GMT bzw. 08. April 12:52:58 CET.
`node_id`	Dieses Attribut dient zur Identifikation, von welchem Solhub System das Event abgeschickt wurde.

	Beispiel: space.fronius.com/sys-182d81d2-a4c5-49e9-a5e5-b1a40397dfc5/container-control/1
`data`	Das Data Objekt ist ein verschachteltes Dokument, was verschiedene Attribute beinhaltet
`channel_id`	Dieses Attribut beinhaltet den Namen bzw. die ID von dem Sensor bzw. dem Bauteil, von welchem das Event ausgelöst wurde.
`event_id`	Die `event_id` gibt an was für ein Event stattgefunden hat. Beispiel: `event_id` 2 ist ein “Water leakage” Event
`severity`	Die `severity` gibt an, wie schwer bzw. schlimm ein Event ist. Dies wird in verschiedenen Stufen unterteilt: 1: fatal; 2: error; 3: warning; 4: info; 5: debug; 6: trace
`message`	Wie der Name schon sagt, gibt diese Attribut die genaue Nachricht was passiert ist. Beispiel: „Water leak detected.“

7.2 Values

Neben den Events gibt es auch die Values. Die Dokumentstruktur der Values ist jedoch etwas komplexer aufgebaut. Das SolHub System sendet, Stand Sommer 2021, Daten von ca. 200 Sensoren. Diese messen beispielsweise den Druck in verschiedenen Bereichen, die Temperatur, die Lüftergeschwindigkeit und vieles mehr. Es werden sekündlich die Daten dieser Sensoren erfasst und in die Datenbank gespeichert.

Abbildung 1: Ausschnitt aus einem Value Dokument in MongoDB Compass

Anstatt für jeden Sensor ein einzelnes Dokument zu erstellen, werden alle Sensoren in ein verschachteltes Dokument eingetragen. Das Dokument besitzt für jeden Sensor ein Array mit dem Namen des Sensors. In diesem befindet sich dann einmal der Wert des Sensors an der ersten Stelle und einmal die Einheit des Sensors an der zweiten Stelle.

8 Filter-Query Language

8.1 Ziel

Die Query-Language soll das einfache Finden von Values und Events in der Datenbank ermöglichen. Hierfür wird eine eigene Sprache benötigt, da die Komplexität mit den verschachtelten Dokumenten und die Menge der Daten, vor allem bei den Values, sehr hoch ist. Dadurch werden handgeschriebene Queries schnell sehr lang, sind umständlich zu schreiben und vor allem für Laien schwer zu verstehen und zu verwenden. Das beste Beispiel hierfür ist die Abfrage der Einheit eines bestimmten Sensors wofür eine Aggregation, wie zum Beispiel die *Value Aggregation* (Siehe Punkt 8.7), verwendet werden muss. Durch die Struktur der Values, wo jeder Sensor ein verschachteltes Array ist, sind diese Abfragen ohne *MongoDB* Aggregationen nur sehr schwer umzusetzen. Das Hauptziel besteht also darin, mit einer simplen Query-Sprache, einem Laien komplexere Abfragen zu ermöglichen, auch wenn wenig Wissen über die Datenstruktur und *MongoDB*-Abfragen vorhanden ist.

8.2 Advanced vs. Basic Variante

Zum einen gibt es im Backend zwei verschiedene Varianten und im Frontend auch noch einmal die Möglichkeit, zwischen Advanced und Basic umzustellen. Zu Beginn war die Advanced-Variante so gedacht, dass sie mehr Funktionen, also mehr Filteroptionen, bereitstellt und so dem

Nutzer mehr Möglichkeiten gibt, die Daten zu filtern. Um auch einem unerfahrenen Nutzer das Filtern zu ermöglichen, sollte es die Basic-Variante geben, welche durch Dropdown-Menüs und andere Features das Zusammenstellen einer Abfrage erleichtern sollte. Wir habe diese Variante über das Frontend so umgesetzt, dass die Daten von Dropdown-Menüs in die Query-Language übersetzt und dann in dieser Form ans Backend übergeben werden.

Die Anforderungen an die zwei Varianten der Sprache haben sich jedoch über die Zeit geändert und dadurch besteht der Unterschied darin das die Advanced Variante Aggregations verwendet und Basic nicht.

Im Backend wurde die Basic-Variante so umfunktioniert, dass sie ohne die sogenannte *Value Aggregation* (Siehe Punkt 8.7), welche für das Filtern von Values verwendet wird, arbeitet und sie wird deswegen nur für Events verwendet. Die Advanced-Variante kann wiederum wird bei der Abfrage von Values verwendet werden und ermöglicht mit der Aggregation das Filtern nach Einheiten und Werten der Sensoren.

Durch den Aufbau der Values (Siehe Punkt 7.2) in der Datenbank ist das Filtern nach Sensorwerten und Einheiten komplex. Das Ganze kann jedoch wieder über die *MongoDB Aggregations Pipeline* (Siehe Punkt 8.6) gelöst werden. Das heißt wiederum, dass das Filtern nach Einheit und Wert eines Sensors nur in der Advanced-Version der Filter-Query möglich ist.

8.3 Definierung der Sprache

Der erste wichtige Schritt ist die Definition der Sprache. Am Anfang war es so gedacht, dass für beide Teile der Query-Language, die Basic und die Advanced-Variante, verschiedene Suchmöglichkeiten existieren sollten. Das Ganze ist zu bestimmten Teilen dazu noch immer im Stande, so kann mit der Advanced Variante bei Values, neben den normalen Möglichkeiten zu filtern, auch nach Werten und Einheiten eines Sensors gefiltert werden. Die Hauptunterschiede sind nun jedoch nicht mehr bei der Sprache selbst sondern bei ihrer Anwendung zu finden.

Um das Suchen nach Events und Values anhand ihrer Properties zu ermöglichen, werden verschiedene Keywords (Siehe Punkt 8.4) verwendet. Diese können unabhängig voneinander und in beliebiger Reihenfolge verwendet werden. Dazu kommt, dass die Sprache für Events und Values ohne Anpassungen funktioniert, solange die Keywords richtig verwendet werden und man sich an die Regeln der Sprache hält.

Das wichtigste Filterkriterium für den Auftraggeber war das Filtern nach Zeit. Hierbei wurde um vier verschiedene Möglichkeiten gebeten. Einmal sollte der Benutzer ein Datum mit oder ohne

Uhrzeit eingeben können, um alle Events oder Values, welche vor diesem Zeitpunkt aufgetreten sind, zu erhalten. Als Gegenstück zu dieser Funktion kann eine Query erstellt werden, um alle Events und Values nach einem bestimmten Zeitpunkt zu erhalten. Neben diesen zwei Optionen kann auch eine Query erstellt werden, um alle Daten zu einem bestimmten Zeitpunkt, oder zwischen zwei Zeitpunkten, zu erhalten.

8.4 Regeln der Sprache

Die Syntax der Query-Sprache wurde sehr einfach gehalten. Grundlegend muss der Benutzer, wenn die Filter-Query-Eingabe wirklich von Hand eingegeben wird, nur ein Keyword und den zu diesem Keyword passenden Parameter verwenden, um eine funktionierende Query zu erhalten. In der nachfolgenden Tabelle sind die Keywords aufgelistet:

Tabelle 3: Keywords der Filter Sprache

Keyword	Beschreibung
`Between`	„Between DD-MM-YYYY HH:mm:ss DD-MM-YYYY HH:mm:ss”: Alle Events/Values zwischen den zwei Zeitpunkten.
`Before`	„Before DD-MM-YYYY HH:mm:ss”: Alle Events/Values vor dem Zeitpunk.t
`After`	„After DD-MM-YYYY HH:mm:ss”: Alle Events/Values nach dem Zeitpunkt.
`At`	„At DD-MM-YYYY HH:mm:ss“: Alle Events/Values welche den genauen Timestamp besitzen.
`With`	Für die Filterung mit den Sub-Keywords.

Als Beispiel könnte man die Query „`Before 19-02-2022`“ verwenden. Das Datum muss hierbei in dem Format „`DD-MM-YYYY`“ geschrieben werden. Neben dem Datum kann optional auch eine Uhrzeit in der Form „`DD-MM-YYYY HH:mm:ss`“ angegeben werden. Neben den Zeit-Keywords gibt es noch das Keyword `with`. Nach diesem muss ein Sub-Keyword folgen, um die Query genauer zu definieren. Hierbei gibt es folgende Sub-Keywords:

Tabelle 4: Sub-Keywords der Filter Sprache

Sub-Keyword	Beschreibung
`unit`	Messeinheit eines Sensors.
`channel_id`	Der Name des Sensors für Event-Queries.
`event_id`	z.B.: Event_id 2 ist ein “Water leakage” event.
`node_id`	Das System welches das Event ausgelöst hat.
`severity`	Die Severity (Siehe Punkt 7.1) eines Events.
`message`	Text den die zusätzliche Nachricht eines Events beinhalten soll.
`sensor`	Der Name des Sensors für Value-Queries.
`value greater/lesser than`	Der Wert eines Sensors.

8.5 Verwendung

Das Verwenden der Filter-Query ist sehr simpel (Siehe Punkt 10.6): Im Frontend wird entweder durch ein Eingabefeld eine Freitext-Eingabe verlangt, welcher der Syntax entspricht, oder durch Date-Picker und ähnliche UI-Elemente die Eingabe vereinfacht. Die daraus entstehende Query wird dann als Query-Parameter in einem REST-Call mitgegeben. Das Backend verwendet dann wiederum die Library, um diesen Query-Parameter zu verarbeiten und eine *MongoDB*-Query zu erhalten. Hierfür wird einfach die Methode `convertStringToQuery` in `filterQuery.js` aufgerufen.

Als Beispiel für eine Query nehmen wir zum Beispiel die Query „Between 01-01-2021 30-12-2021 with unit C and value greater than 20“. Diese wird dann über die in den nächsten Punkten

erklärten Schritte in eine von *MongoDB* verwendbare Form umgewandelt. In der folgenden Abbildung kann man dieses Endergebnis sehen.

```
[
  { '$sort': { _id: -1 } },
  {
    '$addFields': {
      data: {
        '$arrayToObject': {
          '$filter': {
            input: { '$objectToArray': '$data' },
            as: 'el',
            cond: {
              '$and': [
                {
                  '$regexMatch': {
                    input: { '$arrayElemAt': [ '$$el.v', 1 ] },
                    regex: 'c',
                    options: 'i'
                  }
                },
                {
                  '$gt': [ { '$arrayElemAt': [ '$$el.v', 0 ] }, 20 ]
                }
              ]
            }
          }
        }
      }
    }
  },
  {
    '$match': {
      '$and': [ { timestamp: { '$gte': 1609459200, '$lte': 1640822400 } }, {} ]
    }
  },
```

Abbildung 2: Endergebnis des Beispiels

8.6 MongoDB Aggregationen

MongoDB Aggregationen ermöglichen es, eine Query mit mehreren sogenannten *Pipeline Stages* zu erstellen, wobei jede Pipeline eine bestimmte Operation ausführt. Grundlegend können Aggregations mehrere Dokumente einlesen, verarbeiten und danach diese bearbeiteten Ergebnisse zurückgeben. Um Aggregations zu verwenden, kann zum einen die *Aggregation Pipeline*, einzelne Aggregation Methoden, zum anderen *map-reduce operations* verwendet werden. Seit *MongoDB 5.0* sind *map-reduce operations* deprecated (veraltet) und stattdessen soll nur noch die *Aggregation Pipeline* verwendet werden [17].

Ein weiterer sehr vorteilhafter Punkt ist, dass die *Aggregation Pipeline* eine Optimierungsstage hat [18]. Dies kann vor allem bei sehr großen Abfragen hilfreich sein. Schon bei unseren Testdatensätzen von 100.000 Value-Dokumenten, welche jeweils, wie bereits erwähnt, ca. 200 einzelne Values beinhalten, konnten wir hier einen großen Performance-Unterschied, abhängig von den Queries, von bis zu mehreren hundert Millisekunden feststellen.

8.7 Value Aggregation

Die sogenannte *Value Aggregation* ist die wichtigste Aggregation, welche verwendet wird, um die Value-Filterung zu ermöglichen: Sie gibt dem Nutzer die Möglichkeit, nach der Einheit und dem Wert eines Sensors zu filtern und überlässt die Arbeit hierbei der Datenbank.

Abbildung 3: Value Aggregation

Bei der Aggregation, welche für die *Value Aggregation* verwendet wird, handelt es sich um die `$addFields`-Stage. Diese verhält sich ähnlich wie die `$project`-Stage, welche im Backend für das Aufbereiten der Daten verwendet wird. Wir verwenden die `$addFields`-Stage hier dafür, dass nur die benötigten Sensoren, welche den Bedingungen entsprechen, abzufragen.

8.7.1 $addFields Stage

Die `$addFields`-Aggregation-Stage gibt ein Dokument, mit allen existierenden Properties und einem oder mehreren neuen Properties zurück. Die neuen Properties werden einfach an die bereits bestehenden Properties eines Dokumentes angehängt. Seit *MongoDB 4.2* kann statt `$addFields` auch `$set` als alias verwendet werden [19].

Mit dieser Stage können wir also unsere Daten mit der *Value Aggregation* so filtern, dass wir nur die Sensoren, die wirklich benötigt werden, erhalten.

8.7.2 $project Stage

Neben der `$addFields`-Stage verwenden wir auch noch die `$project`-Stage. Sie verhält sich ähnlich wie die `$addFields`-Stage, jedoch müssen bei ihr alle Felder, die ausgegeben werden wollen, angegeben werden. Es werden also nicht alle bisherigen Felder mitgenommen [20].

8.7.3 $match Stage

Diese Stage verwendet normale *MongoDB*-Queries um die Dokumente zu filtern. Hier können also einfach normale Queries, welche von der Filter-Query erstellt wurden, verwendet werden. Hierbei ist es wichtig, die `$match`-Stage als erstes in der Aggregation zu verwenden, um die Verwendung von *MongoDB* Indizes zu ermöglichen, durch welche die Performance gesteigert wird [21].

8.7.4 $limit Stage

Eine weitere Stage ist die sogenannte `$limit`-Stage. Diese Stage gibt eine bestimmte Anzahl an Dokumenten weiter und danach keine mehr. Sie limitiert also wie viele Dokumente an die nächste Pipeline Stage weitergegeben werden sollen [22].

8.7.5 $unwind Stage

Die `$unwind` Stage ist eine der wichtigeren Stages in unserem Programm: Sie wird verwendet, um aus einem Array mit mehreren Objekten ein neues Dokument für jedes Element des Arrays zu erstellen. Wir verwenden sie, um die Sensoren eines Value-Eintrages aufzulösen [23].

8.8 Umsetzung

```
convertStringToQuery: function (toConvert, isAdvanced, offset) {

    //!! This is not very universal and only works for the Flutter frontend
    if (offset !== "") {
        if (!offset.includes("-")) {
            timezoneOffset = "+" + offset.substring(0, offset.lastIndexOf(":"))
        } else {
            timezoneOffset = offset.substring(0, offset.lastIndexOf(":"))
        }
    }

    if (toConvert === "") {
        return []
    }
    if (!isAdvanced) {
        return transformQueryBasic(toConvert);
    } else {
        return transformQueryAdvanced(toConvert);
    }
},
```

Abbildung 4: convertStringToQuery Methode

Wie bereits beschrieben, exportiert `filterQuery.js` die Methode `convertStringToQuery`. Sie ist der Einstiegspunkt in die `filterQuery`. Diese Methode hat drei Parameter, einmal `toConvert`, welcher der String ist, welcher in eine Query umgewandelt werden soll. Neben `toConvert` gibt es noch `isAdvanced`, eine `boolean`-Variable, welche beschreibt, ob es sich um eine Advanced-Query handelt oder nicht. Bei dem letzten Parameter handelt es sich um

offset, welcher für das Beschreiben der Zeitzone vom Client verwendet wird. Dieser wird benötigt, um die Uhrzeiten in die Zeitzone des Servers umzuwandeln.

Nach dem Aufruf der convertStringToQuery-Methode wird entweder die transformQueryBasic- oder transformQueryAdvanced-Methode aufgerufen. Der einzige Unterschied zwischen diesen beiden ist die Verwendung der $addFields-Aggregation für das Filtern nach Einheit oder Wert eines Sensors. Der erste Schritt hier ist das Erstellen des tokenStacks mit der Methode tokenize. Dieser wird im folgenden Abschnitt erklärt.

8.8.1 Tokenizer

```
348 function tokenize(input) {
349     var tokenStack = [];
350 
351     input = input.toLowerCase().trim().replace(",", "");
352     var inputArray = input.split(/\s+/);
353 
354     for (i = 0; i < inputArray.length; i++) {
355         if (isKeyword(inputArray[i])) {
356             tokenStack.push({
357                 type: "Keyword",
358                 value: inputArray[i]
359             })
360         } else if (isSubKeyword(inputArray[i])) {
361             tokenStack.push({
362                 type: "SubKeyword",
363                 value: inputArray[i]
364             })
365         } else if (isDate(inputArray[i])) {
366             var temp = inputArray[i]
367             var needsOffset = false;
368             if (isTime(inputArray[i + 1])) {
369                 //this is the easiest way for adding the time to a date and it working without a
370                 temp += " " + inputArray[i + 1]
371                 i++
372                 needsOffset = true;
373             }
374             tokenStack.push({
375                 type: "Date",
376                 value: temp,
377                 useOffset: needsOffset
378             })
379         } else if (isValueKeyword(inputArray[i])) {
380             //Puts together the "value greater than"/"value lesser than" parts
```

Abbildung 5: Ausschnitt aus der Tokenize methode

Um das Verarbeiten eines Strings zu vereinfachen, wird eine Art Tokenizer verwendet. Dieser splitted den Input zunächst bei allen Whitespaces auf (Siehe Zeile 352, Abbildung 5: Ausschnitt aus der Tokenize methode). Das daraus entstehende Array wird nun in einer For-Schleife durchgegangen, wobei die einzelnen Elemente, abhängig von ihrem Format und ihrem Inhalt, sortiert werden. Sie werden einzeln in den Tokenstack in Form eines Objektes hinzugefügt. Dieses Objekt besitzt zwei Properties oder drei, einmal den type, welcher entweder Keyword, SubKeyword, Date, Severity, Numeric, FreeText sein kann. Neben type wird danach der Wert des Tokens als value in dem Objekt gespeichert. Bei dem letzten Property handelt es sich um useOffset, was bei Date-Tokens benötigt wird, um zu bestimmen, ob das Zeitzonen-Offset verwendet werden muss oder nicht.

8.8.1.1 Utility methoden

Das Sortieren der Tokens wird von verschiedenen Methoden übernommen, welche sich um die Klassifizierung der Tokens kümmern. Sie erhalten alle einen Input `string` und geben ein `boolean` zurück.

Tabelle 5: Methoden der Sprache

Methode	Beschreibung
isDate	Überprüft, ob der `string` ein Datum im Format DD-MM-YYYY ist.
isTime	Überprüft, ob der `string` eine Uhrzeit im Format HH:mm:ss ist.
isKeyword	Überprüft, ob der `string` im keywords Array existiert.
isSubKeyword	Überprüft, ob der `string` im subKeywords Array existiert.
isNumeric	Überprüft, ob der `string` eine Zahl ist.
isSeverity	Überprüft, ob der `string` im Severity Array existiert.
isDateKeyword	Überprüft, ob der `string` „after“, „before“ oder „at“ ist.
isValueKeyword	Überprüft, ob der `string` „than“, „greater“, „lesser“ oder „value“ ist.
isFreeTextOrNumeric	Überprüft, ob der `string` „FreeText“ oder „Numeric“ ist.

8.8.2 Query Erstellung

```
202     for (i = 0; i < tokenStack.length; i++) {
203
204         if (tokenStack[i].type === "Keyword") {
205
206             queryStack.push({})
207             lastToken = tokenStack[i].value;
208
209         } else if (tokenStack[i].type === "SubKeyword") {
210
211             if (lastToken === "with" || lastToken === "and") {
212                 lastToken = tokenStack[i].value;
213             } else {
214                 throw new Error("Found Subkeyword token without 'With' or 'And' token preceding it")
215             }
216
217         } else if (tokenStack[i].type === "Date") {
218             var query = queryStack.pop();
219
220             // Converts the date through regex into a usable format (Month before Day) for the Date constructor
221             var inputDate
222             if (tokenStack[i].useOffset) { // Checks if the Offset is needed or not, if it is used with only a Date then the
223                 inputDate = new Date(tokenStack[i].value.replace(/(\d{2})-(\d{2})-(\d{4})/, "$2/$1/$3") + timezoneOffset)
224             } else {
225                 inputDate = new Date(tokenStack[i].value.replace(/(\d{2})-(\d{2})-(\d{4})/, "$2/$1/$3"))
226             }
227
```

Abbildung 6: Start der Query Erstellung

Nachdem die Tokens erstellt wurden, wird in der Hauptmethode über den sogenannten `tokenStack` mit einer `for`-Loop iteriert. Abhängig von dem Typ des Tokens wird dann eine *MongoDB*-Query erstellt und in den `queryStack` hinzugefügt. Für die Queries selbst werden verschiedene *MongoDB*-Operatoren verwendet, wobei die meisten die Operatoren `$lt`, `$gt` und `$eq` verwenden. Diese bedeuten einmal „lesser than“, „greater than“und „equals“ und filtern die Ergebnisse nach diesen Bedingungen. Am Ende der Methode wird entweder nur der `queryStack` oder auch die *Value Aggregation* (Siehe Punkt 8.7) zurückgegeben.

```
if (lastToken === "between" && !isNaN(inputDate.getTime())) {

    // maybe rework this?
    if (query.timestamp === undefined) {
        query.timestamp = {
            $gte: inputDate.getTime() / 1000
        }
    } else {
        query.timestamp.$lte = inputDate.getTime() / 1000
    }
} else if (isDateKeyword(lastToken) && !isNaN(inputDate.getTime())) {
    if (lastToken === "before") {
        query.timestamp = {
            $lt: inputDate.getTime() / 1000
        }
    } else if (lastToken === "after") {
        query.timestamp = {
            $gt: inputDate.getTime() / 1000
        }
    } else if (lastToken === "at") {
        query.timestamp = {
            $eq: inputDate.getTime() / 1000
        }
    }
} else {
    throw new Error("Unexpected Date token: " + tokenStack[i].value)
```

Abbildung 7: Time Query Erstellung

8.9 Tests

Für die Filter-Query wurden verschiedenste Tests geschrieben, um zu beweisen, dass sie auch wirklich funktioniert. Um dies zu erreichen, verwenden wir das Backend und führen verschiedene Requests mit den gewünschten Filtern aus, welche wir überprüfen wollen. Mit diesen Tests stellen wir sicher, dass unsere Abfragen die richtigen Ergebnisse liefern.

```
15  test("GET /values with unit C and greater than 20", async () => {
16
17      await supertest(app).get("/values?filter=With unit C and value greater than 20&advanced=true")
18          .expect(200)
19          .then((response) => {
20              expect(response.body.length).toBe(1)
21              expect(response.body[0].unit).toBe("C");
22              expect(response.body[0].sensorvalue).toBeGreaterThan(20)
23          });
24  });
```

Abbildung 8: Ausschnitt aus den Tests

9 Backend

Das Backend bietet über REST-Anfragen die Möglichkeit für das Frontend, sich die benötigten Daten zu holen. Hierfür werden verschiedene Requests für Values und Events bereitgestellt, um die verschiedenen Anforderungen des Frontends zu erfüllen. Es wurde in NodeJS mithilfe von Express erstellt.

Ein weiterer wichtiger Punkt ist die Performance des Backends. Schon bei den derzeitigen Testmengen an Daten, 100 000 Datensätze and Events und Values, werden diese im Backend so gut wie möglich für das Frontend aufbereitet, um ein schnelles Laden zu ermöglichen. Dies geschieht durch die sogenannte *valueProjection.* Sie erstellt bei einer Abfrage für jeden Sensor ein eigenes, Dokument.


```
{
    "_id":"6070669197a69d085314c3fb",
    "node_id":"space.fronius.com/sys-182d81d2-a4c5-49e9-a5e5-b1a40397dfc5/container-control/1",
    "timestamp":1617978870,"sensorname":"coc/acely_tt102_ai",
    "sensorvalue":23.62700080871582,
    "unit":"C"
}
```


Abbildung 9: Sensor Dokument nach der valueProjection

9.1 Backend Architektur

Das Backend wurde zunächst in einzelne Teile aufgeteilt. Nach Best-Practices für NodeJS wurden hierfür zunächst Services, Router und Controller für jeden der zwei Datentypen erstellt. Im Service werden die Daten aus der Datenbank geholt und wieder an das Frontend zurückgeschickt. Der Router ist für die Verarbeitung der URL und das Weiterleiten zuständig. Im Controller werden dann die Parameter aus der URL und dem Body des Requests geholt und die dazugehörige Methode aus dem Services ausgeführt. Zu guter Letzt kommt ein Teil, welcher dafür zuständig ist, die Verbindung zur Datenbank herzustellen. Hierfür wird der *MongoDB-NodeJS-Driver* verwendet.

9.2 Datenaufbereitung

Zu Beginn wurden alle Daten in ihrer Originalform aus der Datenbank, also unaufbereitet, an das Frontend geschickt und dort verarbeitet. Dieser Prozess war langsam und ineffizient da die Daten komplett im Frontend verarbeitet werden mussten. Nach etwas Recherchearbeit haben wir uns dafür entschieden, die Daten so weit wie möglich im Backend aufzubereiten. Hierfür werden die unter Punkte 8.6, genannten *MongoDB*-Aggregationen verwendet, um vor allem die 200 Values in einzelne Dokumente aufzuteilen.

Das Aufbereiten wird hierbei von der sogenannten *valueProjection* erledigt. Bei der *valueProjection* handelt es sich wie schon beschrieben um eine *MongoDB*-Aggregation, bestehend aus der `$project`-Stage.

```
const valueProjection = {
    "$project": {
        "data": {
            "$objectToArray": "$data"
        },
        "timestamp": 1,
        "node_id": 1
    }
}
```

Abbildung 10: Value Projection

Die Projection erstellt mit `$objectToArray` ein neues Dokument mit einem Array aus den Sub-Dokumenten eines Value Dokumentes als data Attribut. Nach dieser Projection wird noch die `$undwind`-Stage (Siehe Punkt 8.7.5) verwendet, um jedes Element in dem mit `$objectToArray` neuerstellten Array zu einem neuen Dokument umzuwandeln. Danach werden diese Dokumente noch einmal mit einem `foreach` zusammengefasst, um die immer noch verschachtelten Werte und Einheiten der Sensoren für das Frontend leichter zugänglich zu machen.

9.2.1 DbHelper

Das Backend verwendet zum Verbinden mit der Datenbank den *MongoDB* Node.JS Driver. Die Verbindung selbst kann über die Methode `connect` im File *dbHelper.js* erstellt werden. Neben `connect` bietet *dbHelper.js* auch noch eine `close` und eine `get`-Methode: Die `close` Methode schließt die Verbindung mit der Datenbank und die `get` Methode gibt die Datenbank-Connection zurück.

```
1  const mongoClient = require('mongodb').MongoClient;
2
3  const mongoDbUrl = 'mongodb://127.0.0.1:27017';
4  const dbName = "solhub"
5  let mongodb;
6
7  async function connect(callback) {
8      await mongoClient.connect(mongoDbUrl, {
9          useNewUrlParser: true,
10         useUnifiedTopology: true
11     }, (err, client) => {
12         mongodb = client;
13         callback();
14     });
15 }
16
17 function get() {
18     return mongodb.db(dbName);
19 }
20
21 function close() {
22     mongodb.close();
23 }
24
25 module.exports = {
26     connect,
27     get,
28     close
29 };
```

Abbildung 11: DbHelper.js

9.2.2 Router

Es gibt im Backend jeweils einen Router für Events und einen für Values. Diese verwenden die bestehende Router-Middleware von Express.js (Siehe Punkt 6.5.2) um http-Requests den richtigen Methoden im Controller zu verweisen.

```
var express = require('express');
var router = express.Router();

var EventController = require('../controllers/eventController')

router.get('/', EventController.getEvents)

router.get('/After/:ObjectId', EventController.getEventsAfter)

router.get('/Before/:ObjectId', EventController.getEventsBefore)

router.get('/Predict',EventController.predictEvent)

module.exports = router;
```

Abbildung 12: eventRouter.js

9.2.3 Controller

Ähnlich wie beim Router gibt es wieder einen Controller für Events und einen für Values. In beiden gibt es für jeden HTTP-Request eine Methode, in welcher die benötigten Parameter aus der URL und dem Request Body entnommen werden. Danach wird mit diesen Parametern die richtige Methode im Service aufgerufen.

```
const EventService = require('../services/eventService')
const fs = require('fs');

exports.getEvents = async function (req, res, next) {
    // Validate request parameters, queries using express-validator

    var documentLimit = req.query.limit ? Number.parseInt(req.query.limit) : 50;
    var filter = req.query.filter ? req.query.filter : "";
    var offset = req.query.timezoneOffset ? req.query.timezoneOffset : "";

    var result = await EventService.getEvents(documentLimit, filter, offset).catch(next)
    res.send(result)
}

exports.getEventsAfter = async function (req, res, next) {
    // Validate request parameters, queries using express-validator

    var objectId = req.params.ObjectId ? req.params.ObjectId : -1
    var documentLimit = req.query.limit ? Number.parseInt(req.query.limit) : 50;
    var filter = req.query.filter ? req.query.filter : "";
    var offset = req.query.timezoneOffset ? req.query.timezoneOffset : "";

    var result = await EventService.getEventsAfter(documentLimit, filter, objectId, offset).catch(next)
    res.send(result)
}

exports.getEventsBefore = async function (req, res, next) {
    // Validate request parameters, queries using express-validator

    var objectId = req.params.ObjectId ? req.params.ObjectId : -1
    var documentLimit = req.query.limit ? Number.parseInt(req.query.limit) : 50;
    var filter = req.query.filter ? req.query.filter : "";
    var offset = req.query.timezoneOffset ? req.query.timezoneOffset : "";

    var result = await EventService.getEventsBefore(documentLimit, filter, objectId, offset).catch(next)
    res.send(result)
}
```

Abbildung 13: Ausschnitt aus eventController.js

9.2.4 Service

Wie bei den vorherigen zwei Teilen gibt es ebenso zwei Services, einen für Events und einen für Values. Die Services kümmern sich um das Beschaffen der Daten aus der Datenbank. Hierfür wird wieder der *dbHelper* verwendet, um sich mit der bereits erwähnten `get`-Methode die Datenbank Verbindung zu holen.

9.3 Requests

9.3.1 Values

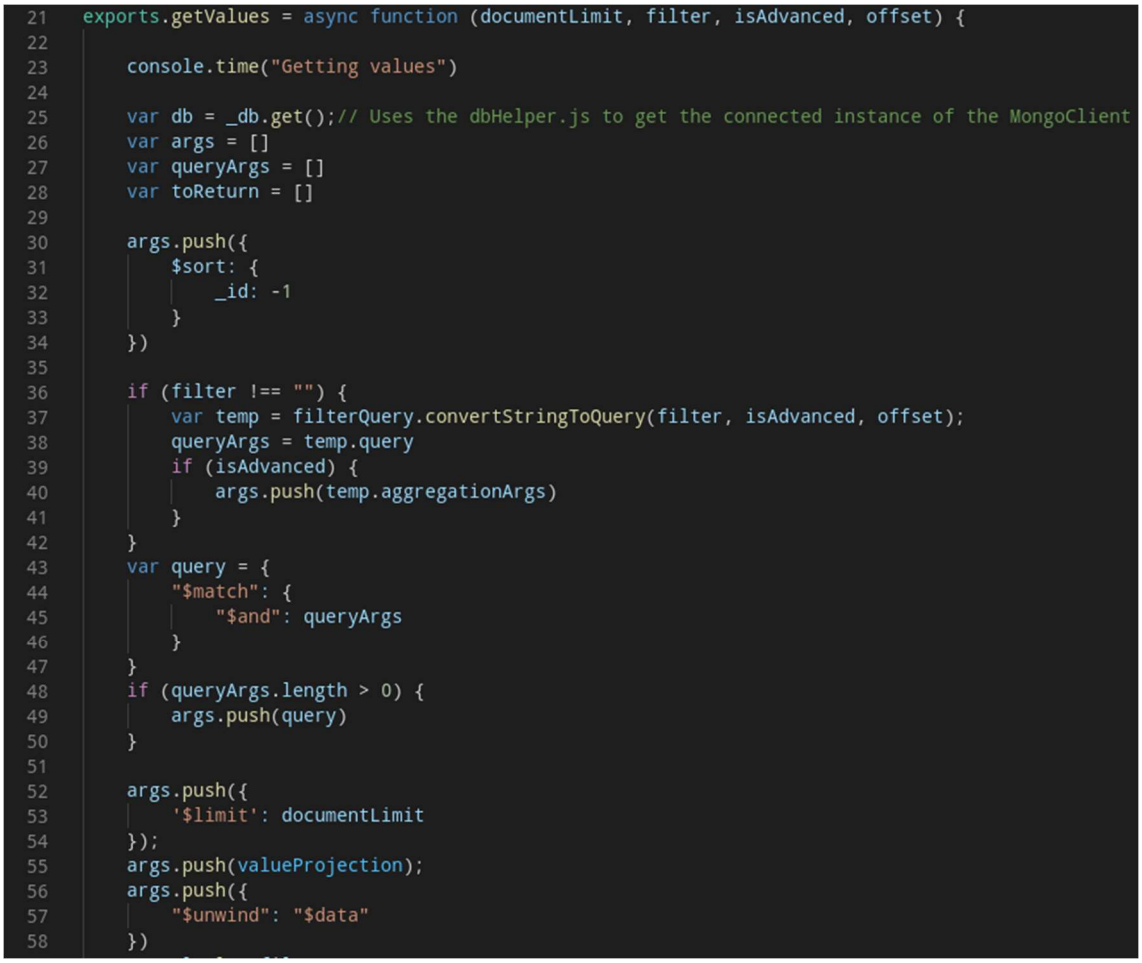
In den folgenden Punkten werden, die in der Abbildung zu sehenden einzelnen Requests für Values genauer erklärt.

```
router.get('/', ValueController.getValues)

router.get('/After/:ObjectId', ValueController.getValuesAfter)

router.get('/Before/:ObjectId', ValueController.getValuesBefore)

router.get('/Sensor/:ObjectId', ValueController.getValuesForSensor)

router.get('/Event/:ObjectId', ValueController.getValuesForEvent)

router.get('/Predict', ValueController.getValuePredictionForSensor)
```

Abbildung 14: Value Router

9.3.1.1 getValues

Der erste Request nimmt hierbei die Parameter `filter`, `isAdvanced`, `offset` und `limit` an und gibt die, nach Timestamp sortierten, ersten 200 Sensoren Dokumente zurück. Bei `filter` handelt es sich um die Filter-Query die verwendet werden soll. *IsAdvanced* ist dafür da, wenn eine Advanced-Query, also mit Aggregationen, benötigt wird. `Offset` ist der Timezone-Offset, wie bereits in der Filter-Query erklärt. Das Limit ist standardmäßig auf ein Dokument gesetzt, da in einem Dokument ca. 200 Sensoren vorhanden sind.



```
exports.getValues = async function (documentLimit, filter, isAdvanced, offset) {

    console.time("Getting values")

    var db = _db.get();// Uses the dbHelper.js to get the connected instance of the MongoClient
    var args = []
    var queryArgs = []
    var toReturn = []

    args.push({
        $sort: {
            _id: -1
        }
    })

    if (filter !== "") {
        var temp = filterQuery.convertStringToQuery(filter, isAdvanced, offset);
        queryArgs = temp.query
        if (isAdvanced) {
            args.push(temp.aggregationArgs)
        }
    }
    var query = {
        "$match": {
            "$and": queryArgs
        }
    }
    if (queryArgs.length > 0) {
        args.push(query)
    }

    args.push({
        '$limit': documentLimit
    });
    args.push(valueProjection);
    args.push({
        "$unwind": "$data"
    })
```

Abbildung 15: Ausschnitt aus getValues Methode

9.3.1.2 getValuesAfter/getValuesBefore

Neben dem normalen `getValues` Request gibt es auch noch Requests für `getValuesAfter` und `getValuesBefore`. Diese haben neben den gleichen Parametern, welche `getValues` besitzt auch, noch `objectId` als Parameter. Mit diesem Parameter geben sie die Values, welche nach dem mit `objectId` definierten Value Dokument oder vor dem mit `objectId` bestimmten Value Dokument sind, zurück.

9.3.1.3 getValuesForSensor

Ein weiterer Request ist `getValuesForSensor`. Er wird dazu verwendet, um die Values von einem Sensor, anhand seines Namens, in einem bestimmten Zeitraum nach dem mit `objectId` definierten Value-Dokument zu erhalten. Die Parameter unterscheiden sich hierbei von den vorherigen Requests: Neben der ObjectId gibt es den `sensorName`, welcher den Sensor, der gesucht wird, definiert und das `timeLimit`, welches den Zeitraum definiert. Dieser ist standardmäßig zehn Minuten, was bedeutet, dass nach Werten zehn Minuten vor und nach dem mit `objectId` definierten Dokument gesucht wird.

9.3.1.4 getValuesForEvent

Neben `getValuesForSensor` gibt es auch `getValuesForEvent`. Dieses Request verhält sich sehr ähnlich zu `getValuesForSensor`, statt nach einem Sensor zu suchen, werden jedoch die Events zwei Minuten vor und nach einem übergebenen Event gesucht.

9.3.1.5 getValuePredictionForSensor

Das letzte Request bei Values ist `getValuePredictionForSensor`, welches dafür verwendet wird, um die eine Vorhersage für den Wert eines Sensors mit Predictive Maintenance (Siehe Punkt 11.2.4) zu machen. Der Sensor wird hierbei mit dem Parameter `sensorName` definiert.

```
346
347     // spawn new child process to call the python script
348     var python = spawn('python3',
349     [
350         `${process.cwd()}/../predictive_maintenance/Prediction.py`, // Name of the python script
351         selectedSensor.data.k, // Severity of the event to predict
352         selectedSensor.data.v[0] // The current/last known value of that sensor
353     ]);
354
355     python.stderr.pipe(process.stderr) // Pipes the python error console logs into node.js console
356
357     python.on('close', (code)=>{
358         console.log(`Python exited with code: ${code}`)
359         callBack(selectedSensor.data.v[1] )
360     });
```

Abbildung 16: Ausschnitt aus getValuePredictionForSensor

9.3.2 Events

In den folgenden Punkten werden, die in der Abbildung zu sehenden einzelnen Requests für Events genauer erklärt.

```
router.get('/', EventController.getEvents)

router.get('/After/:ObjectId', EventController.getEventsAfter)

router.get('/Before/:ObjectId', EventController.getEventsBefore)

router.get('/Predict',EventController.predictEvent)
```

Abbildung 17: Event Router

9.3.2.1 getEvents

Ähnlich wie bei Values dient der `getEvents`-Request dazu, um die ersten 50 Events abzufragen. Hierfür werden die Parameter `filter`, `timezoneOffset` und `limit` benötigt. Die Verwendung dieser Parameter unterscheidet sich hier nicht von den Value-Requests.

```
8   exports.getEvents = async function (documentLimit, filter, offset) {
9
10      console.time("Getting events")
11      var db = _db.get();
12      var cursor;
13
14      var queryArgs = filterQuery.convertStringToQuery(filter, false, offset).query;
15
16      var query = {
17          "$and": queryArgs
18      }
19
20      if (filter !== "") {
21          cursor = db.collection('log/events').find(query)
22      } else {
23          cursor = db.collection('log/events').find()
24      }
25
26      var toReturn = await cursor.sort({
27          _id: -1
28      }).limit(documentLimit).toArray();
29
30      console.timeEnd("Getting events")
31
32      return toReturn
33  };
```

Abbildung 18: getEvents Methode

9.3.2.2 getEventsAfter/getEventsBefore

Nach `getEvents` gibt es auch wieder `getEventsBefore` und `getEventsAfter`. Dieser Request gibt entweder die 50 Events, die direkt nach oder vor dem mit ObjectId definierten Event gespeichert sind. Die restlichen Parameter neben ObjectId sind wieder gleich wie bei `getEvents` und den Value-Requests.

9.3.2.3 predictEvent

Das `predictEvent`-Request ist wiederum dafür da, den Predictive Maintenance (Siehe Punkt 11.2.4) zu benutzen. Hierfür wird nur ein Parameter, toPredict, benötigt. Dieser sagt aus, welche Severity das Event, das vorhergesagt wird, haben soll. Mit diesem wird dann das *Python*-File ausgeführt und das Ergebnis zurückgegeben.

10 Frontend

10.1 Überblick

Das Frontend ist einfach aufgebaut:

Home: Wie der Name schon sagt, ist das unsere Startseite, wo der User als erstes landet, wenn er unsere Seite aufmacht. Hier gibt es keine Funktionalität.

Live View: Die Live View war keine Anforderung im Umfang unserer Diplomarbeit und hat daher ebenso keine Funktionalität.

Log Events: Die Log-Events-Seite ist dafür zuständig, dem User die zeittechnisch neusten Events in einer tabellarischen Ansicht anzuzeigen. Dort kann der User dann nach spezifischen Events filtern und durch das Anklicken eines Event Eintrags einen Graphen erstellen.

Log Values: Die Log-Values-Seite ist, ähnlich wie die Log-Events-Seite, dafür zuständig die neuesten Values in einer tabellarischen Ansicht anzuzeigen. Auch hier kann man dann nach genaueren Values filtern und durch das Anklicken eines Value-Eintrags einen Graphen erstellen.

Graph View: Der Graph View selbst macht nichts, da er zuerst einen Eintrag, entweder Event oder Value, braucht, um davon einen Graphen zu erstellen. Sobald ein Eintrag angeklickt wird, wird der User zum Graph View weitergeleitet und der Graph wird erstellt.

Predictions: Die Predictions-Seite ist dafür zuständig, Predictions im Frontend auszuführen. Hier gibt es verschiedene Buttons für die jeweils verschiedenen Sensoren und auch Events,die vorhergesagt werden können.

10.2 Navigation

Die Navigation selbst wird über die `layout_template` Klasse realisiert; hier bauen wir die Seite zusammen. Da die Navigation immer gleich bleibt, ändert sich immer nur der inhaltliche Teil der Webseite. Wenn man die Seite startet, landet man immer auf der Home-Route.

```dart
class LayoutTemplate extends StatelessWidget {
  const LayoutTemplate({Key? key}) : super(key: key);

  //Draws the Home Page
  @override
  Widget build(BuildContext context) {
    return ResponsiveBuilder(
        builder: (context, sizingInformation) => Scaffold(
            drawer:
                sizingInformation.deviceScreenType == DeviceScreenType.mobile
                    ? const NavigationDrawer()
                    : null,
            backgroundColor: Colors.white,
            body: Column(children: <Widget>[
              const NavigationBar(),
              Expanded(
                child: Navigator(
                  key: locator<NavigationService>().navigatorKey,
                  onGenerateRoute: generateRoute,
                  initialRoute: HomeRoute,
                ), // Navigator
              ) // Expanded
            ]))); // <Widget>[] // Column // Scaffold // ResponsiveBuilder
  }
```

Abbildung 19: build Methode von layoutTemplate

Im Code sieht man den `Navigator`, der dafür zuständig ist, uns zu den jeweiligen Seiten zu navigieren und die richtigen Informationen anzuzeigen [24] [25]. Dieser braucht dann unseren `NavigatorService`.

```dart
//Simple Navigation Service to navigate through the different Sites/Views
class NavigationService {
  final GlobalKey<NavigatorState> navigatorKey = GlobalKey<NavigatorState>();

  Future<dynamic> navigateTo(String routeName) {
    return navigatorKey.currentState!.pushNamed(routeName);
  }

  void goBack() {
    navigatorKey.currentState!.pop();
  }
}
```

Abbildung 20: NavigationService Klasse

Die Route-Namen selbst sind in einem eigenen File festgelegt, da sie immer gleichbleiben:

```
const String HomeRoute = 'home';
const String LiveViewRoute = 'liveView';
const String LogEventsRoute = 'logEvents';
const String LogValuesRoute = 'logValues';
const String GraphViewRoute = 'graphView';
const String PredictionRoute = 'predictionView';
```

Abbildung 21: Routen

Das jetzt jedoch die Route auch funktioniert und sie uns dorthin weiterleitet, wo wir hin wollen müssen wir in unserem Router die jeweiligen Konstanten/Routen auch generieren und angeben, wohin weitergeleitet werden soll. Dies machen wir in unserer `generateRoute` Methode:

```
Route<dynamic> generateRoute(RouteSettings settings) {
  switch (settings.name) {
    case HomeRoute:
      return _getPageRoute(const HomeView());
    case LiveViewRoute:
      return _getPageRoute(const HomeView());
    case LogValuesRoute:
      return _getPageRoute(const ValueView());
    case LogEventsRoute:
      return _getPageRoute(const EventView());
    case GraphViewRoute:
      return _getPageRoute(const GraphView(
        id: "no Arguments",
        isEvent: true,
        date: "",
      ));
    case PredictionRoute:
      return _getPageRoute(const PredictionView());
    default:
      return _getPageRoute(const HomeView());
  }
}

PageRoute _getPageRoute(Widget child) {
  return _FadeRoute(child: child);
}
```

Abbildung 22: generateRoute Methode

Die FadeRoute ist auch eine eigene Klasse, die nur für die Transition bzw. die Animation zuständig ist:

```
class _FadeRoute extends PageRouteBuilder {
  final Widget child;
  _FadeRoute({required this.child})
      : super(
            pageBuilder: (BuildContext context, Animation<double> animation,
                    Animation<double> secondaryAnimation) =>
                child,
            transitionsBuilder: (
              BuildContext context,
              Animation<double> animation,
              Animation<double> secondaryAnimation,
              Widget child,
            ) =>
                FadeTransition(
                  opacity: animation,
                  child: child,
                ));
}
```

Abbildung 23: FadeRoute Klasse

10.3 Home

Die Home-Seite ist sehr einfach gestaltet und dient nur zur Begrüßung des Users und zur Navigation zu den jeweilig anderen Seiten.

Abbildung 24: Startseite

10.4 Live View

Die Live View war keine Anforderung im Umfang unserer Diplomarbeit und stellt daher keine Funktionalität bereit.

10.5 Log Events

Log Events, wie schon der Name verrät, ist dafür da, die Events anzuzeigen. Dies geschieht in einer Tabellenansicht. Dazu machen wir einen Backend-Call, holen uns alle benötigten Daten und zeigen diese an. Danach kann der Benutzer über verschiedene Filter (oder in der Advanced-

Ansicht über Query-Filter), die Daten eingrenzen, die in der Tabelle angezeigt werden. Filtert der User nicht, wird die Tabelle stetig neu geladen, um immer die neuesten Daten anzuzeigen. Falls der User den Filter nicht mehr benutzen will, kann er auf „Reset“ drücken, um wieder die neuesten Daten anzuzeigen.

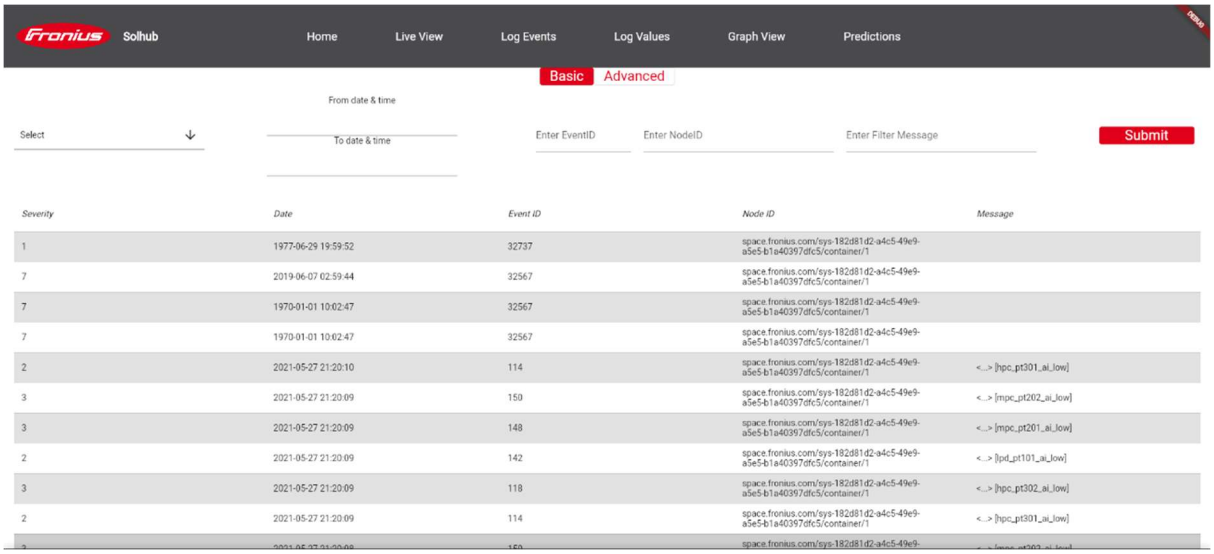


Abbildung 25: Log Events Ansicht

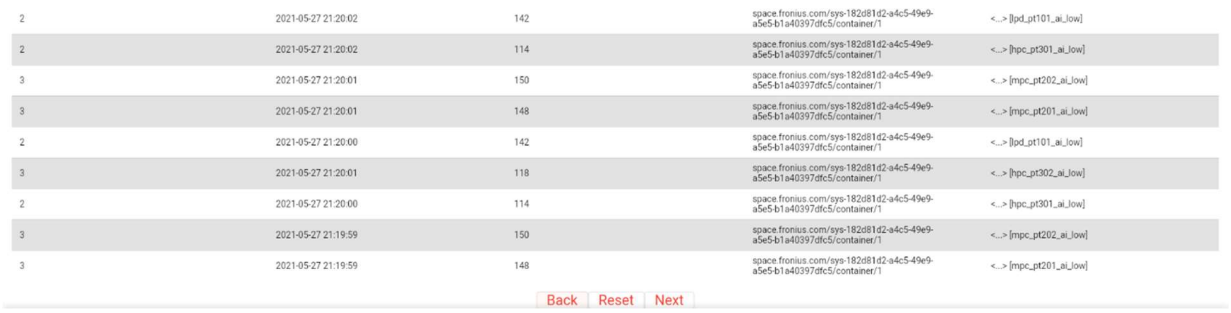


Abbildung 26: Seitenende Log Events

Die Advanced View, wie schon oben erwähnt, benutzt eine Text Query als Filter, wodurch es möglich ist, noch genauer zu filtern. Ein Beispiel wäre „With severity error and event_id 77“. Mehr zu der Filtersprache kann bei Punkt 8 nachgelesen werden.

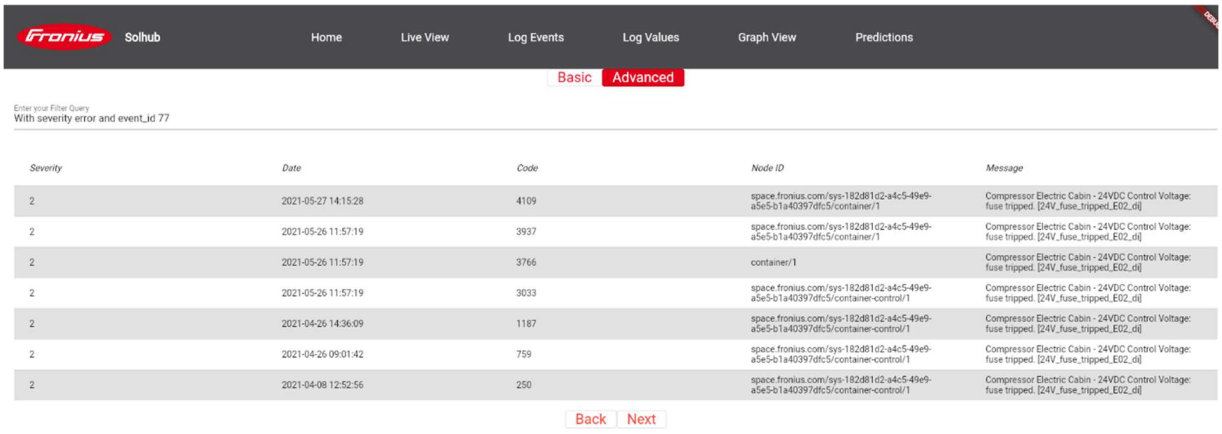


Abbildung 27: Advanced Ansicht Log Events

Die Daten selbst können angeklickt werden, was dann dazu führt, dass wir zur Graph View weitergeleitet werden. Dort wird dann der Zeitverlauf zwei Minuten vor bzw. nach dem jeweiligen Event angezeigt wird. Die Daten selbst sind Werte von dem jeweiligen Sensor bei dem das Event aufgetreten ist. Das heißt, wenn ein Event ausgelöst wird, beispielsweise von einem Temperatursensor, werden dann die jeweiligen Temperaturen zwei Minuten vorher und nachher angezeigt. Über den Graphen selbst gibt es bei Punkt 10.8 mehr zu lesen.

10.6 Log Values

Ähnlich wie die Log-Events-Seite, zeigt die Log-Values-Seite die Value-Daten in einer Tabelle an. Das Filtern ist fast gleich wie bei den Events jedoch gibt es andere Filter, passend zu den Values. Statt der `severity` und der `eventID`, kann man hier nur nach `node_id`, `channel_id` und Datum filtern:

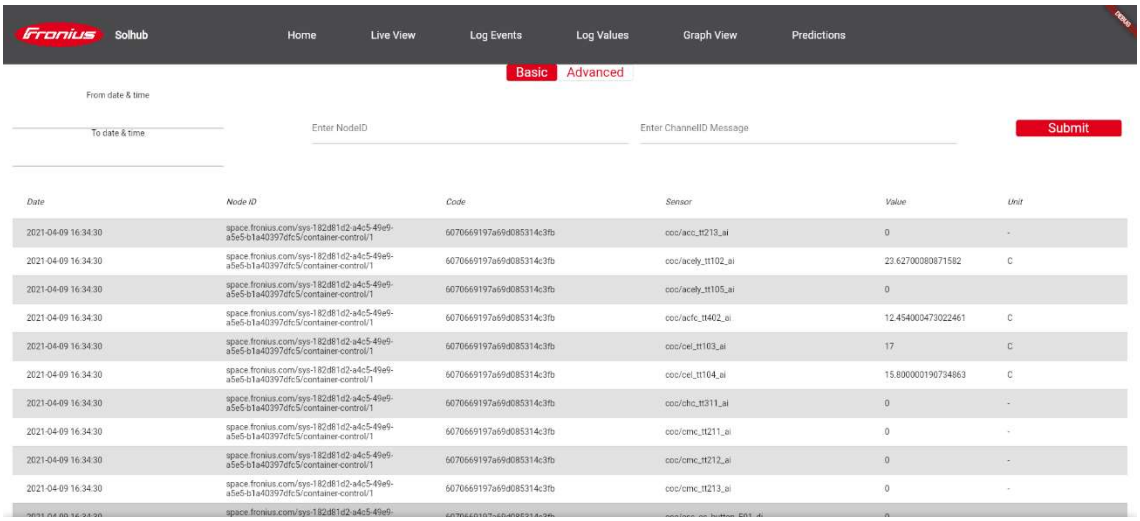


Abbildung 28: Log Values Ansicht

In der Advanced View, ähnlich wie bei der Advanced View auf der Event-Seite, kann man natürlich spezifischer werden, wobei hier jedoch zwei Filtermöglichkeiten dazukommen: Unit und Value. Eine Beispiel Query, wo wir diese zwei Attribute verwenden, wäre „Before 08-04-2021 12:52:57 With unit C and value lesser than 15“. Wenn der User die Query syntaktisch falsch eingibt oder etwas Unsinniges, wird ihm eine Fehlermeldung angezeigt. Diese lautet: „Oops something went wrong. Maybe your Filter Query was wrong“. Mehr zu der Filtersprache kann bei Punkt 8 gelesen werden.

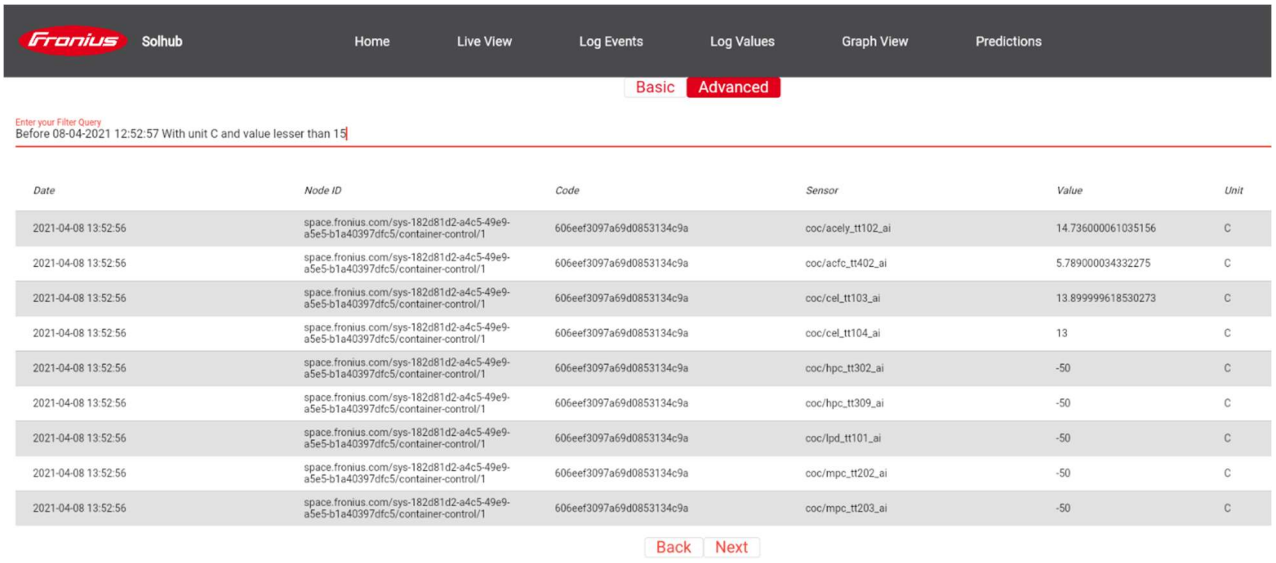


Abbildung 29: Advanced Ansicht Log Values

Wie bei der Log Events Seite auch, wird man beim Anklicken eines Eintrags in der Tabelle zur Graph View weitergeleitet. Dort wird dann, im Gegensatz zu den Events, der Zeitverlauf von zehn Minuten vor bzw. nach dem jeweiligen Value angezeigt.

10.7 Aufrufen der Daten und Verarbeitung für Tabellen

10.7.1 Daten Aufruf ohne Filter

Damit die Seiten wissen, was und welche Daten sie anzeigen müssen, wird ein Backend-Call gemacht. Dieser ist für die Log-Values-Seite und Log-Events Seite gleich, der einzige Unterschied ist nur welcher URL benutzt wird (statt localhost:3001/events -> localhost:3001/values).

```dart
@override
void initState() {
  //The first initialization; Creates the timer and sends a Get of all Events to the Backend
  super.initState();
  futureEvents = fetchEvent("0");
  reloadTimer = Timer.periodic(const Duration(seconds: 5), (timer) {
    if (usingTimer) {
      futureEvents = fetchEvent("0");
      setState(() {});
    }
  }); // Timer.periodic
}
```

Abbildung 30: initState Methode

Im ersten Screenshot sehen wir die `initState` Methode, die aufgerufen wird, sobald die Log-Events-Seite oder Log-Values-Seite aufgemacht wird. Dort machen wir den Call zur `fetchEvent`-Methode mit Wert „0“, damit wir wissen, dass es der erste Aufruf ist. Auch sehen wir den `reloadTimer`, der alle fünf Sekunden die `fetchEvent`-Methode aufruft, um immer die neusten Daten anzuzeigen. Hier haben wir den `usingTimer`-Check, der, wie der Variablenname schon sagt, dafür da ist, um zu wissen, ob wir den Timer benutzen oder nicht. Der Timer wird nicht mehr gebraucht, sobald der User den „Next“ oder „Back“ Knopf drückt.

```dart
Future<List<Event>> fetchEvent(String hexnum) async {
  //gets the data from backend
  BigInt num = BigInt.parse(hexnum, radix: 16);
  final http.Response response;
```

Abbildung 31: fetchEvent Methode

```dart
  //Checks what Event needs to be sent:
  //0: new Page is opened or it is reloaded because of the timer
  //!isBack: calls the after call to get the next page; is used by the After Button
  //isBack: calls the before call to get the page before; is used by the Back Button
  //if all else fails, the else is called for the standard procedeure
  if (num == BigInt.from(0)) {
    usingTimer = true;
    isBack = false;
    response = await http.get(Uri.parse('http://localhost:3001/events'));
    //Between 08-04-2021 10:52:50 08-04-2021 10:53:00
  } else if (!isBack) {
    reloadTimer.cancel();
    usingTimer = false;
    isBack = false;
    response = await http
        .get(Uri.parse('http://localhost:3001/events/before/' + hexnum));
  } else if (isBack) {
    reloadTimer.cancel();
    usingTimer = false;
    isBack = true;
    response = await http
        .get(Uri.parse('http://localhost:3001/events/after/' + hexnum));
  } else {
    usingTimer = true;
    isBack = false;
    response = await http.get(Uri.parse('http://localhost:3001/events'));
  }
```

Abbildung 32: Überprüfung des Nutzerinputs

In diesen Screenshots sehen wir die `fetchEvent`-Methode, die für den Backend-Call zuständig ist und dafür, die erhaltenen Daten weiterzuleiten. Als allererste machen wir den check, ob der User das erste Mal die Seite aufruft oder ob er auf den Knopf „Back“ oder „Next“ gedrückt hat. Hierfür checken wir die umgewandelte `hexnum`, was ein `timestamp` ist. Diesen brauchen wir wenn der User nach vorne oder zurück geht, damit das Backend weiß, welcher Eintrag der letzte war, damit es die richtigen nächsten Einträge schicken kann. Wie schon erwähnt, wenn es der erste Aufruf ist, wird die `hexnum` immer „0“ sein. Wie ebenfalls bereits gesagt, wird der Timer beendet, sobald der User „Back“ oder „Next“ drückt, da sonst immer weiter aktuelle Daten geladen werden, obwohl man historische Daten haben will.

```dart
if (response.statusCode == 200) {
  final Iterable i = json.decode(response.body);

  //checks if it was a before or after call, so the List can be sorted properly
  if (i.isNotEmpty && !isBack) {
    //Uses the Mapped data to create a List;Check event_dto.dart for more information on .fromJson
    List<Event> events =
        List<Event>.from(i.map((ev) => Event.fromJson(ev)));
    return events;
  } else if (i.isNotEmpty && isBack) {
    List<Event> events = List<Event>.from(i.map((ev) => Event.fromJson(ev)))
        .reversed
        .toList();
    return events;
  } else {
    throw Exception('The List that was returned was empty');
```

Abbildung 33: JSON Konvertierung und Verarbeitung

Nachdem wir nun den Call ins Backend gemacht haben, bekommen wir eine JSON zurück, dass wir in eine Liste umwandeln, um damit arbeiten zu können. Zu den Requests selbst ist mehr unter Punkt 9.3 erklärt.

Falls der User auf den „Back“-Knopf gedrückt hat, kommen die Daten verkehrtherum an, also müssen wir diese umdrehen, damit sie dann richtig in der Tabelle angezeigt werden. Nachdem wir das alles gemacht haben, geben wir die Liste zurück, mit der wir dann arbeiten.

10.7.2 Datenaufruf mit Filter

Natürlich müssen wir die Filter auch berücksichtigen, wenn der User diese benutzt. Sobald der User den Knopf „Submit“ drückt, kommen wir in die `fetchQueryEvent`-Methode.

```dart
Future<List<Event>> fetchQueryEvent(String hexnum) async {
  final http.Response response;

  //Builds the String for the Query, to be sent to the Backend
  String sev = "";
  String query = "";

  if (first) {
    //Checks if the Severity Picker is empty
    if (selectedSev.isNotEmpty) {
      //if it isn't, it combines all Selected severities into one String
      sev = "With severity ";
      selectedSev.forEach((val) => sev += "${val!.name} ");
      query += sev;
    }

    //Checks if the filter of eventId is empty; Adds it to the query
    if (eventId.isNotEmpty) {
      query += "With event_id $eventId ";
    }
    //Same as eventId
    if (nodeid.isNotEmpty) {
      query += "With node_id $nodeid ";
    }
    //Same as eventId
    if (message.isNotEmpty) {
      query += "With message $message ";
    }
```

Abbildung 34: fetchQueryEvent Methode

Die ersten Checks sind dafür da, um zu überprüfen, ob der User einen der Filter ausgewählt bzw. etwas hineingeschrieben hat. Da der `severity`-Filter ein `MultiSelectDialogField` [26] ist (eine Art ComboBox in Flutter), setzen wir die `selectedSev` mit den Werten aus dem Dialog, die ausgewählt worden sind. Durch diese iterieren wir dann durch und fügen sie zu unserem `query`-String hinzu. Danach checken wir jeweils für `event_id`, `node_id` und `message`, ob etwas hineingeschrieben worden ist. Wenn der check `true` ist, fügen wir auch diese zu unserem `query`-String hinzu.

```dart
    //Checks if Dates were entered and which ones
    if (fromDate.isNotEmpty && toDate.isNotEmpty) {
      //if a From and To Date were entered we need to use the Keyword "Between"
      query = "Between $fromDate $toDate $query";
    } else if (fromDate.isNotEmpty && toDate.isEmpty) {
      //if a just a From Date was entered we need to use the Keyword "After"
      query = "After $fromDate $query";
    } else if (fromDate.isEmpty && toDate.isNotEmpty) {
      //if a just a To Date was entered we need to use the Keyword "After"
      query = "Before $toDate $query";
    }
    //Puts the built query into savedQuery so it can be used afterwards if the User presses "Next" or "Back"
    savedQuery = query;
  }

  //Rest of the code is same as in Event View Advanced; Check there for more infos on this part of the code (check Method "fetchQueryEvent")
  if (first) {
    response = await http.get(Uri.parse(
        'http://localhost:3001/events?filter=$savedQuery&timezoneOffset=${DateTime.now().timeZoneOffset}'));
    first = false;
  } else if (!isBack) {
    response = await http.get(Uri.parse(
        'http://localhost:3001/events/before/$hexnum?filter=$savedQuery&timezoneOffset=${DateTime.now().timeZoneOffset}'));
  } else if (isBack) {
    response = await http.get(Uri.parse(
        'http://localhost:3001/events/after/$hexnum?filter=$savedQuery&timezoneOffset=${DateTime.now().timeZoneOffset}'));
  } else {
    response = await http.get(Uri.parse('http://localhost:3001/events'));
  }
```

Abbildung 35: Datumsüberprüfung und Backend Call

Als nächstes überprüfen wir die Eingabe des Datums. Da es einen Unterschied macht, ob der User nur ein „From“- oder nur ein „To“-Datum eingegeben hat, oder auch beides, müssen wir überprüfen, welcher von diesen Fällen eintritt. Wie auch schon in Punkt 8.4 beschrieben, werden verschiedene Keywords für die Filter-Query gebraucht, um richtig nach Datum zu filtern.

```
Future<List<Sensor>> fetchQueryEvent(String hexnum) async {
  final http.Response response;

  String query = "";

  if (first) {
    if (nodeid.isNotEmpty) {
      query += "With node_id $nodeid ";
    }
    if (channelId.isNotEmpty) {
      query += "With sensor $channelId&advanced=true";
    }

    if (fromDate.isNotEmpty && toDate.isNotEmpty) {
      query = "Between $fromDate $toDate $query";
    } else if (fromDate.isNotEmpty && toDate.isEmpty) {
      query = "After $fromDate $query";
    } else if (fromDate.isEmpty && toDate.isNotEmpty) {
      query = "Before $toDate $query";
    }

    savedQuery = query;
  }
```

Abbildung 36: fetchQueryEvent für Values

Der einzige Unterschied zwischen der Log-Events- und der Log-Values-Seite ist, dass die URL anders ist und dass es keine `message`, `severity` und `event_id` gibt, sondern nur `node_id` und `channel_id` (Siehe 7.1 und 7.2).

10.7.3 Datenverarbeitung

Nachdem wir nun die Daten geholt haben, müssen wir diese weiterverarbeiten. Um die Daten auch verwenden zu können, benutzen wir den sogenannten `FutureBuilder` von Flutter. Dieser wartet, bis unsere Future-Variable Daten hat und baut dann daraus die Tabelle aus [27].

```
204     //Future Builder is used so that Data that is continuously read can be displayed
205     return FutureBuilder(
206         future: futureEvents,
207         builder: (BuildContext context, AsyncSnapshot<List<Event>> snapshot) {
208           //Checks if the Data "arrived" from the Backend
209 >         if (snapshot.hasData) {···
519           } else if (snapshot.hasError) {
520             return Text("${snapshot.error}");
521           } else {
522             return const Center(
523                 child: Text("Loading....",
524                     style: TextStyle(
525                       fontSize: 35,
526                     ))); // TextStyle // Text // Center
527           }
528         }); // FutureBuilder
```

Abbildung 37: FutureBuilder in EventAnsicht

Das Erstellen der Tabelle geschieht wie folgt:

```
Padding(
  padding: const EdgeInsets.all(16),
  //creates the Table
  child: DataTable2(
    horizontalMargin: 12,
    columns: const <DataColumn>[
      DataColumn2(
        label: Text(
          'Severity',
          style:
              TextStyle(fontStyle: FontStyle.italic),
        ), // Text
      ), // DataColumn2
      DataColumn2(
        label: Text(
          'Date',
          style:
              TextStyle(fontStyle: FontStyle.italic),
        ), // Text
      ), // DataColumn2
      DataColumn2(
        label: Text(
          'Event ID',
          style:
              TextStyle(fontStyle: FontStyle.italic),
        ), // Text
      ), // DataColumn2
```

Abbildung 38: Spaltennamen Generierung Teil 1

```
      DataColumn2(
        label: Text(
          'Node ID',
          style:
              TextStyle(fontStyle: FontStyle.italic),
        ), // Text
      ), // DataColumn2
      DataColumn2(
        label: Text(
          'Message',
          style:
              TextStyle(fontStyle: FontStyle.italic),
        ), // Text
      ), // DataColumn2
    ], // <DataColumn>[]
```

Abbildung 39: Spaltennamen Generierung Teil 2

Wie der Widget Name schon aussagt, haben wir ein DataTable2-Widget, wobei die zwei für eine Erweiterung steht. Es hat mehr Funktionen als `DataTable` [28]. Dort fügen wir unsere Spalten hinzu und geben ihnen jeweils einen Namen (`label`).

```
rows: List<DataRow2>.generate(
  events.length,
  (int index) => DataRow2(
    //Adds OnTap for every Table entry;Used to create Graph
    onTap: () {
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) => GraphView(
                    id: events[index].id,
                    isEvent: true,
                    date: events[index].date,
                  ))); // GraphView // MaterialPageRoute
    },
    color:
        MaterialStateProperty.resolveWith<Color?>(
            (Set<MaterialState> states) {
      // All rows will have the same selected color.
      if (states
          .contains(MaterialState.selected)) {
        return Theme.of(context)
            .colorScheme
            .primary
            .withOpacity(0.08);
      }
```

Abbildung 40: Row Generierung

```
            // Even rows will have a grey color.
            if (index.isEven) {
              return Colors.grey.withOpacity(0.3);
            }
            return null; // Use default value for other states and odd rows.
          }),
```

Abbildung 41: Odd/Even Überprüfung

Hier generieren wir unsere Reihen. Das `generate` von `List<DataRow2>` gibt uns die Möglichkeit, einen `index` mitzugeben, den wir auch durchiterieren können. Nachdem wir das gemacht haben, geben wir dem `generate` die Länge unserer Liste mit, die mit Events gefüllt ist. Als nächstes geben wir jedem Eintrag eine Funktion mit, die ausgeführt wird, sobald der User diesen Eintrag anklickt. Diese Funktion ist dafür da, dass wir für den jeweiligen Eintrag zur Graph View übergeleitet werden. Wir wissen welcher Eintrag dies ist, da wir den `index` haben, was bereits erwähnt worden ist. Danach geben wir den Datensätzen eine Hintergrundfarbe. Wir geben auch mit, dass jeder zweiter Eintrag einen anderen Grauton bekommt.

```
        //Adds the Data to the Cell
        cells: <DataCell>[
          DataCell(Text(events[index]
              .data
              .severity
              .toString())), // Text // DataCell
          DataCell(
              Text(events[index].date.toString())), // DataCell
          DataCell(Text(
              events[index].data.eventId.toString())), // Text // DataCell
          DataCell(
              Text(events[index].nodeId.toString())), // DataCell
          DataCell(Text(
              events[index].data.message.toString())), // Text // DataCell
        ], // <DataCell>[]
      ), // DataRow2
    ), // List.generate
  ), // DataTable2
), // Padding
```

Abbildung 42: Daten Generierung für Cells

Nachdem wir das alles gemacht haben, geben wir nun jeder Reihe ihre Daten. Dafür erstellen wir pro Spalte ein `DataCell`-Objekt und befüllen dieses mit den Daten des jeweiligen Eintrages in der Liste [29]. Hierfür brauchen wir wieder den Index, um zu wissen bei welchem Eintrag wir zurzeit sind.

10.8 Graph View

Die Graph View, direkt über das Menü ausgewählt, hat keine Funktionalität. Wenn der User diese Seite aufrufen will, wird er informiert, dass er zuerst einen Datensatz auswählen soll.

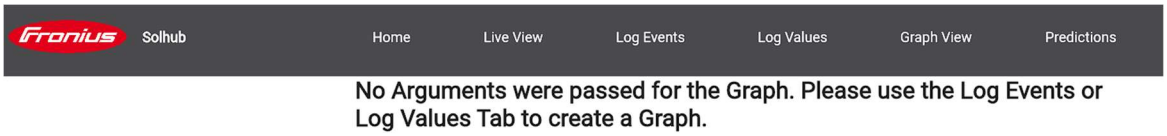


Abbildung 43: Graph View ohne Arguments

10.8.1 Event

Wie schon in Punkt 10.5 und 10.6 erwähnt, wird man durch das Klicken auf einen Event-Datensatz zur Graphenansicht geschickt.

Abbildung 44: Event Graph

Wie wir hier in dem Beispiel sehen, gab es einen Zeitsprung und direkt danach kam es zu dem Event, dass die Temperatur zu niedrig ist. Dieser Zeitsprung entstand durch einen Ausfall/Neustart des Systems.

10.8.2 Value

Wie schon in Punkt 10.5 und 10.6 erwähnt, wird man durch das Klicken auf einen Value-Datensatz zur Graphenansicht geschickt.

Abbildung 45: Value Graph

Wie man im obigen Screenshot sieht, werden die letzten zehn Minuten angezeigt, ab dem Moment, den der Datensatz beschreibt. In dem Beispiel kann man auslesen, dass die Temperatur des Sensors bzw. Bauteils von 12,24 C° auf 12,44 C° gestiegen ist.

10.8.3 Datenaufrufen

Wie schon bei Punkt 10.7.3 besprochen, geben wir beim Klick a auf einen Tabelleneintrag Daten mit und leiten den Benutzer zur Graph View weiter. Die Daten selbst sind die `id` vom Event bzw. vom Value, ein `boolean` um überprüfen zu können, ob es ein Event oder Value Eintrag war, der Sensorname (falls es ein Value Eintrag war) und das Datum an dem das Event bzw. der Value aufgetreten ist.

```
Future<List<Sensor>> fetchGraphData() async {
  late final http.Response response;

  //checks if the call was sent from the Events View or Values View
  if (widget.isEvent) {
    response = await http
        .get(Uri.parse('http://localhost:3001/values/event/${widget.id}'));
  } else {
    response = await http.get(Uri.parse(
        'http://localhost:3001/values/sensor/${widget.id}?sensorname=${widget.sensorname!.split("/")[1]}'));
  }

  final Iterable i = json.decode(response.body);

  //checks if it was a before or after call, so the List can be sorted properly
  if (i.isNotEmpty) {
    //Uses the Mapped data to create a List;Check event_dto.dart for more information on .fromJson
    List<Sensor> events =
        List<Sensor>.from(i.map((ev) => Sensor.fromJson(ev)));
    return events;
  } else {
    throw Exception('No Data provided');
  }
}
```

Abbildung 46: fethGraphData-Methode

Natürlich müssen wir auch die Graphdaten vom Backend holen. Dies machen wir in unserer `fetchGraphData`. Als erstes machen wir den Check, ob es ein Event oder Value war, von dem der Call kam. Der Call unterscheidet sich darin, dass man für den Value Eintrag auch den Sensornamen braucht.

10.8.4 Datenverarbeitung

Wie bei der Datenverarbeitung bei der Tabelle in 10.7.3, brauchen wir auch hier den `FutureBuilder`, um die Daten benutzen zu können. Für den Graphen selbst benutzen wir `charts` von Flutter selbst, genauer gesagt den `TimeSeriesChart`, da dieser alle Funktionen hat, die wir benötigen [30].

```
if (snapshot.hasData) {
  List<Sensor> values = snapshot.data as List<Sensor>;
  seriesList = _createSampleData(values);
  //Checks if the Graph will be Drawn from an Event or a Sensor point of view
  if (widget.isEvent) {
    return charts.TimeSeriesChart(
      seriesList,
      animate: false,
```

Abbildung 47: Überprüfung ob Daten vorhanden sind

Bevor wir jedoch die Daten mitgeben, müssen wir die Daten zu einer `Series` konvertieren, da der Graph nur `Series` zum Anzeigen akzeptiert [31] [32]. `Series` ist eine Gruppe von Informationen, die wir zum Zeichnen unseres Graphens brauchen [33]. Dies machen wir in der `_createSampleData` Methode, der wir unsere Values mitgeben.

```
static List<charts.Series<Sensor, DateTime>> _createSampleData(
    List<Sensor> val) {
  final data = val;

  return [
    charts.Series<Sensor, DateTime>(
      id: val[0].nodeId.toString(),
      colorFn: (_, __) => charts.MaterialPalette.cyan.shadeDefault,
      domainFn: (Sensor sensor, _) => DateTime.parse(sensor.date),
      measureFn: (Sensor sensor, _) => sensor.val,
      data: data,
    )
  ];
}
```

Abbildung 48: _createSampleData-Methode

Die Klasse `Series` hat zwei Datentypen, einmal für die X-Achse und einmal für die Y-Achse: Die X-Achse ist die Zeit und die Y-Achse sind die Messwerte unseres Sensors. Nun erstellen wir die Liste mit den Daten, die wir mitgegeben haben. Die ID ist für jeden Wert gleich, da es stets der gleiche Sensor ist, genauso die Farbe. Die `domainFn` (also die X-Achse) ist, wie schon erwähnt, unsere Zeit. Das heißt wir geben von jedem Eintrag das `timestamp` mit bzw. wandeln diese um. Die `measureFn` (oder auch Y-Achse) ist der Messwert des Sensors, zum jeweiligen Zeitpunkt. Nachdem wir das alles gemacht haben, geben wir die Liste zurück. Nun können wir unseren Graphen erstellen bzw. angeben, wie er ausschauen und wie er sich verhalten soll. Dies machen wir in den `behaviours`.

```
return charts.TimeSeriesChart(
  seriesList,
  animate: false,

  behaviors: [
    charts.LinePointHighlighter(
      showHorizontalFollowLine:
          charts.LinePointHighlighterFollowLineType.nearest,
      showVerticalFollowLine:
          charts.LinePointHighlighterFollowLineType.nearest,
      symbolRenderer: CustomCircleSymbolRenderer(),
    ), // charts.LinePointHighlighter
```

Abbildung 49: Graph Erstellung

Als erstes geben wir an, dass beim ausgewählten Wert zwei orthogonal zueinanderstehenden Linien erstellt werden sollen. Dies sieht man hier:

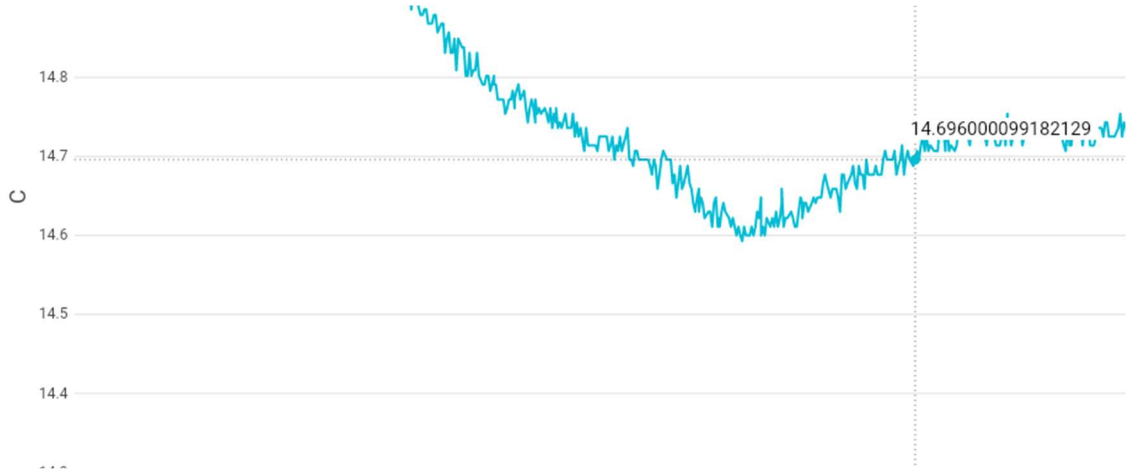


Abbildung 50: Hilfslinien für ausgewählten Wert

```
    // By default, select nearest is configured to trigger
    // with tap so that a user can have pan/zoom behavior and line point
    // highlighter. Changing the trigger to tap and drag allows the
    // highlighter to follow the dragging gesture but it is not
    // recommended to be used when pan/zoom behavior is enabled.
    charts.SelectNearest(
        eventTrigger: charts.SelectionTrigger.tap), // charts.SelectNearest
    //creates a Vertical line to indicate at what Time the Event was triggered
    charts.RangeAnnotation([
      charts.LineAnnotationSegment(DateTime.parse(widget.date),
          charts.RangeAnnotationAxisType.domain,
          startLabel: 'Event'), // charts.LineAnnotationSegment
    ]), // charts.RangeAnnotation
```

Abbildung 51: Erstellung Bereichsanmerkung

Auch geben wir mit, dass immer der ausgewählte Wert angezeigt wird. Danach erstellen wir eine vertikale Linie an der Stelle, wo das Event bzw. der ausgewählte Value-Eintrag stattgefunden hat:

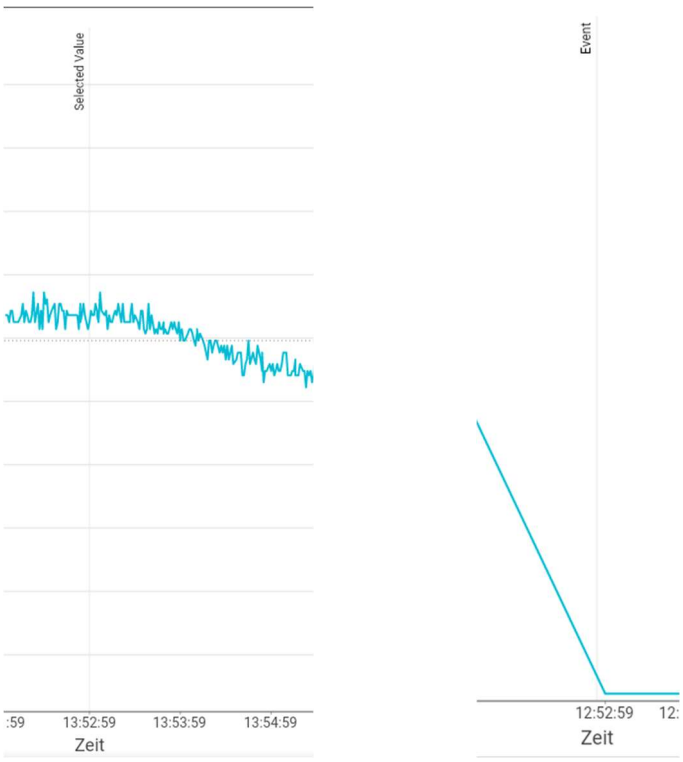


Abbildung 52: Bereichsanmerkung

```
  //Sets Title of the Axis; Indicates what Units are used; X-Axis will always be Time, but can be changed
  charts.ChartTitle('Zeit',
      behaviorPosition: charts.BehaviorPosition.bottom,
      titleOutsideJustification:
          charts.OutsideJustification.middleDrawArea), // charts.ChartTitle
  charts.ChartTitle(values.first.unit.toString(),
      behaviorPosition: charts.BehaviorPosition.start,
      titleOutsideJustification:
          charts.OutsideJustification.middleDrawArea), // charts.ChartTitle
],
```

Abbildung 53: Erstellung Achsenbeschriftung

Als nächstes geben wir unseren Achsen einen Titel. Die X- Achse ist immer Zeit, die Y-Achse jedoch ist immer die jeweilige Maßeinheit des Sensors.

```
//Sets what the primaryMeasure Axis is (y-Axis)
primaryMeasureAxis: charts.NumericAxisSpec(
  renderSpec: const charts.SmallTickRendererSpec(
      // Change the line colors to match text color.
      lineStyle: charts.LineStyleSpec(
          color: charts.MaterialPalette.black)), // charts.LineStyleSpec // charts.SmallTickRendererSpec
  //Sets the Ticks aka the Range Indicators for the Axis
  tickProviderSpec: const charts.BasicNumericTickProviderSpec(
    zeroBound: false,
    dataIsInWholeNumbers: false,
  ), // charts.BasicNumericTickProviderSpec
```

Abbildung 54: Generierung Y-Achse

```
//Sets from and to for the Axis aka the viewport
viewport: charts.NumericExtents(
    values
            .reduce((value, element) =>
                value.val < element.val ? value : element)
            .val -
        0.15,
    values
            .reduce((value, element) =>
                value.val > element.val ? value : element)
            .val +
        0.15), // charts.NumericExtents
), // charts.NumericAxisSpec
```

Abbildung 55: Bereich der Y-Achse erstellen

Danach setzen wir unsere `primaryMeasureAxis` (die Y-Achse). Hier geben wir den `tickProviderSpec` an, der dafür zuständig ist, die Achsenbeschriftung zu erstellen. Danach setzen wir auch den `viewport` der Achse, der die Achsenlänge definiert. Hierfür haben wir einmal „from“, welches den ersten Wert nimmt und dann minus 0.15 abzieht, damit die Tabelle nicht sofort beim ersten Wert beginnt. Für das „to“ machen wir das Gegenteil davon. Hier nehmen wir den letzten Wert und fügen 0.15 dazu, damit die Achse nicht beim letzten Wert er Tabelle endet.

```
//Sets the domain Axis (x-Axis); in our case it will always be time so it's a DateTimeAxis
domainAxis: charts.DateTimeAxisSpec(
    tickFormatterSpec: charts.BasicDateTimeTickFormatterSpec(
        (datetime) =>
            DateFormat('HH:mm:ss').format(datetime)), // charts.BasicDateTimeTickFormatterSpec
    tickProviderSpec:
        //Creates the Time Indicators for the Axis; Check Method for more info
        charts.StaticDateTimeTickProviderSpec(
            _createTickSpecList(values)), // charts.StaticDateTimeTickProviderSpec
```

Abbildung 56: Generierung X-Achse

Danach machen wir das gleiche für die `domainAxis` (die X-Achse), die unsere Zeitachse ist. Hierfür brauchen wir eine eigene Methode, die die Zeitwerte in eine `TickSpecList` umwandelt, dass sie der `StaticDateTimeTickProviderSpec` verwenden kann. Diese schaut wie folgt aus:

```
List<charts.TickSpec<DateTime>> _createTickSpecList(List<Sensor> values) {
  List<charts.TickSpec<DateTime>> tickSpecList = [];
  DateTime start = DateTime.parse(values.first.date);
  DateTime end = DateTime.parse(values.last.date);
  DateTime current = start;
  tickSpecList.add(charts.TickSpec(start,
      label: values.first.date.split(" ")[1],
      style: const charts.TextStyleSpec(fontSize: 14))); // charts.TickSpec

  //Checks if the TickSpecs are for Event or Sensor Graph;
  //Mostly the same, the only key difference is the time range in which the Ticks are set
  if (widget.isEvent) {
    values.forEach((val) => {
          if (DateTime.parse(val.date).millisecondsSinceEpoch >=
                  current.millisecondsSinceEpoch + 10000 &&
              DateTime.parse(val.date).microsecondsSinceEpoch <
                  end.microsecondsSinceEpoch)
            {
              tickSpecList.add(charts.TickSpec(DateTime.parse(val.date),
                  label: val.date.split(" ")[1],
                  style: const charts.TextStyleSpec(fontSize: 14))), // charts.TickSpec
              current = DateTime.parse(val.date)
            }
        });
```

Abbildung 57: _createTickSpecList-Methode

Hier holen wir uns jeweils den Startzeit und die Endzeit. Danach fügen wir den Anfang in unsere `tickSpecList` ein. Dann prüfen wir wieder, ob die Datensätze von der Seite Log Events oder Log Values kommen, da die Zeitspannen vor und nach dem Event/Value verschieden sind. Danach kontrollieren wir, ob der `current`-Wert, der am Anfang `start` ist, zwischen Anfang und Ende liegt. Nachdem wir das erfolgreich überprüft haben, iterieren wir durch unsere Liste von Werten, wandeln die Zeit in ein `DateTime` Datentyp um und fügen diese dann in unsere `tickSpecList` ein. Nachdem wir das alles gemacht haben, geben wir diese Liste zurück. Damit kann der `StaticDateTimeTickProviderSpec` arbeiten und die Zeitachse richtig zeichnen.

```
      //Set the min and max value of the vertical line
      viewport: charts.DateTimeExtents(
          start: DateTime.parse(values.first.date),
          end: DateTime.parse(values.last.date))), // charts.DateTimeExtents // charts.DateTimeAxisSpec
```

Abbildung 58: Bereich der X-Achse erstellen

Zu guter Letzt setzen wir auch für die `domainAxis` den `viewport`, was in unserem Fall ganz einfach das Datum des ersten Werts und das Datum des letzten Werts in der Liste ist.

```
//Sets the Selection Model (changedListener, OnClick etc.)
selectionModels: [
  charts.SelectionModelConfig(
    //We only have a Changed Listener at the moment,
    //which creates a small "bubble" above the clicked point to show the Value at that point in time
    changedListener: (charts.SelectionModel model) {
      if (model.hasDatumSelection) {
        final val = model.selectedSeries[0]
            .measureFn(model.selectedDatum[0].index)
            .toString();
        CustomCircleSymbolRenderer.value =
            val; // paints the tapped value
      }
    },
  ) // charts.SelectionModelConfig
],
); // charts.TimeSeriesChart
```

Abbildung 59: Setzen der selectionModels

Als letztes setzen wir die sogenannten `selectionModels`. Hier erstellen wir einen `SelectionModelConfig` und geben diesem einen `changedListener` mit, der dafür zuständig ist ein Textfeld zu zeichnen, um den Wert des ausgewählten Datensatzes im Graphen anzuzeigen. Das sehen wir hier:

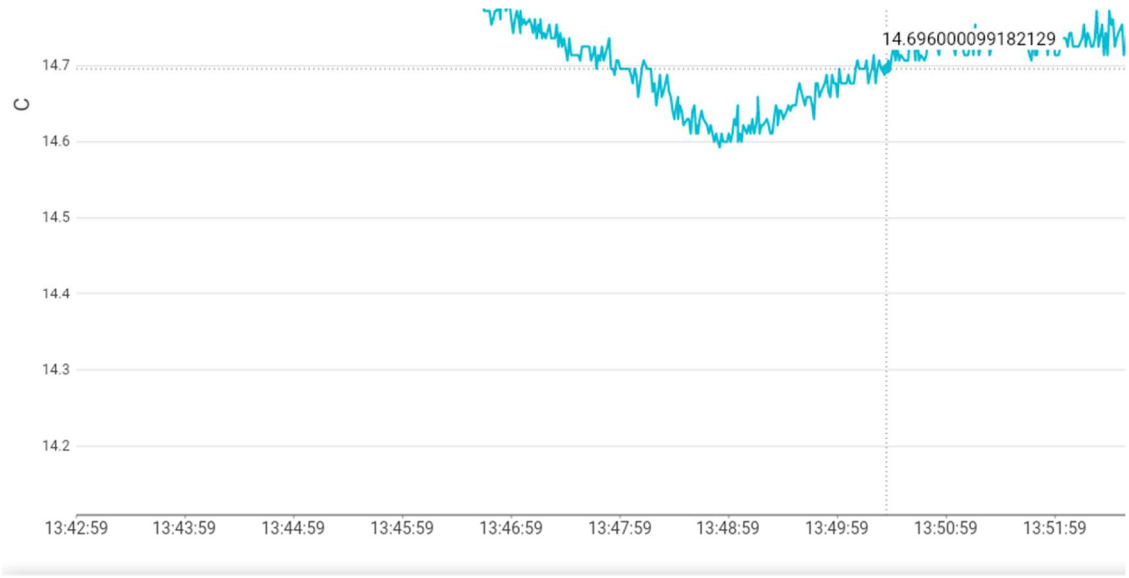


Abbildung 60: Bubble mit Wert des ausgewählten Eintrags

10.9 Predictions

Die Predictions Seite ist dafür zuständig, einen Backend-Call zu machen, der die Predictions ausführt und das Ergebnis zurückschickt. Dieses Ergebnis wird dann im Frontend angezeigt. Es gibt Knöpfe für die jeweiligen Sensoren (Siehe Punkt 11.2.5). Beim Event-Knopf, wird das nächste Event vorhergesagt, mit der `severity` zwei (Error).

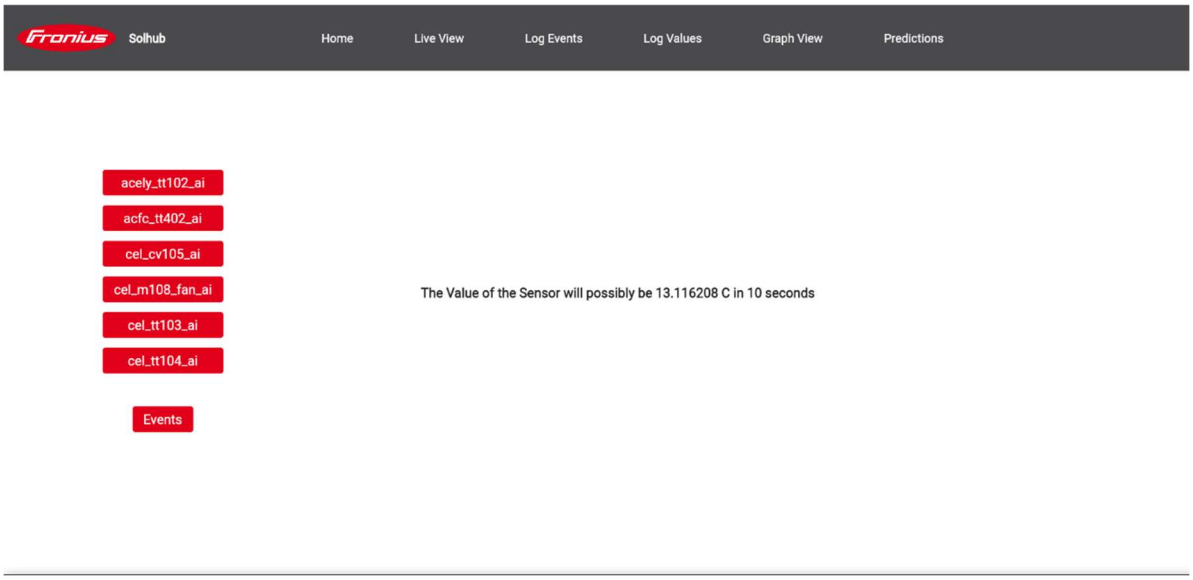


Abbildung 61: Predictions Seite

11 Predictive Maintenance

Predictive Maintenance ist ein weitreichender Begriff. In unserem Fall ist damit, die Vorhersage von Werten der Sensoren (Values), aufgrund historischer Daten und derzeit verfügbaren Daten, sowie die Vorhersage der Zeitdauer bis zu einem nächsten *Event* gemeint. Konkret bedeutet das, dass mit Hilfe der Messwerte verschiedener Sensoren in Bar, Volt, etc. über eine Woche zunächst ein System antrainiert wurde. Dieses System ist dann dazu in der Lage Vorhersagen zum Sensorwert in den nächsten zehn Sekunden zu treffen. Ähnliches gilt für die *Events*. Um beides zu tun, benötigt es einen Input in Form des derzeitigen Wertes dieses Sensors.

11.1 Theorie

11.1.1 Begrifflichkeiten

Künstliche Intelligenz oder englisch *Artificial Intelligence (AI)* meint jegliche Art von scheinbarer Intelligenz, die ein Computersystem in der Lage ist vorzugeben und dem Bewältigen von intellektuellen (logischen Denken erforderlichen) Aufgaben dient [34, p. 4]. „Scheinbar“ und „vorgeben“ werden deshalb so verwendet, weil man noch nicht von einer Intelligenz im Sinne des menschlichen Gehirnes sprechen kann. Zu dieser Definition gehören beispielsweise bereits simple Computergegner in Spielen, die nur mit einfachen Abfragen arbeiten, was *Symbolic AI* genannt wird [34, p. 4], aber auch fortgeschrittenere Konzepte wie *Machine Learning*.

Machine Learning kann als eine Art Umkehrung des herkömmlichen Programmierens gesehen werden. Während man beim Programmieren die Regeln (die Logik) und die Daten vorgibt und

Resultate erwartet, werden beim *Machine Learning* die Daten und die Resultate verwendet, um daraus die Regeln zu lernen. Dabei werden die Daten in verschiedene, für den Computer verständliche, Repräsentation konvertiert. *Deep Learning* knüpft daran an und erhöht die Anzahl der internen Repräsentationen erheblich [34, pp. 5-14].

11.1.2 Das Model und seine Bestandteile

Einem solchen *Deep Learning* Systems, wie es hier entstehen soll, liegt das sogenannte *Model* zugrunde. Das *Model* ist das endgültige System, das nach dem Training und der damit einhergehenden Konfiguration in der Lage sein soll, neue Daten entgegenzunehmen und daraus Ergebnisse (in diesem Fall Vorhersagen) abzuleiten.

Das *Model* besteht aus verschiedenen *Layers*. Diese werden durch drei Metriken bestimmt: die *Activation Function* (oder dt. *Aktivierungsfunktion*), die Anzahl der *Input/Output Units* (oder künstliche Neuronen) und der Art ihrer Verbindung:

- Die *Aktivierungsfunktion* bestimmt, wie die Werte transformiert werden. Sie liefert einen groben Anhaltspunkt, wie die Werte verändert werden. Eine *Relu-Aktivierungsfunktion* (kurz für *rectified linear unit*) beispielsweise setzt alle negativen Werte auf null [34, p. 87]. Wichtige Einflussgrößen dieser Funktionen sind die *Weights*. Sie ermöglichen die feingradige Anpassung, des Resultats der Funktionen and die erwarteten Resultate, um den Fehler zu minimieren.
- Die *Input/Output Units* sind eine Stellschraube, die indirekt beeinflussen, wie „kreativ" das System beim Lernen sein darf. Eine große Zahl ermöglicht dem System sich auch kleinere Nuancen in Daten zu merken/erlernen, während eine kleinere es dazu zwingt sich auf das Nötigste zu beschränken [34, p. 100].
- Die Art der Verbindung bestimmt, wie die *Layers* miteinander verbunden werden. Es gibt einige verschiedene Arten, aber für dieses Beispiel sind nur die *Densely Connected* oder kurz *Dense* Verbindungen interessant.

Die nachstehenden Grafiken veranschaulichen den erklärten Sachverhalt noch einmal auf einen Blick. Die erste Grafik erklärt den Zusammenhang zwischen den *Weights* (Gewichtungen) und den verschiedenen Funktionen, wobei die *Übertragungsfunktion* außer Acht gelassen werden kann, da sie in diesem Fall nur die Summe der Werte aller *Input/Output Units* berechnet. Sie wurde bezogen von Quelle [35].

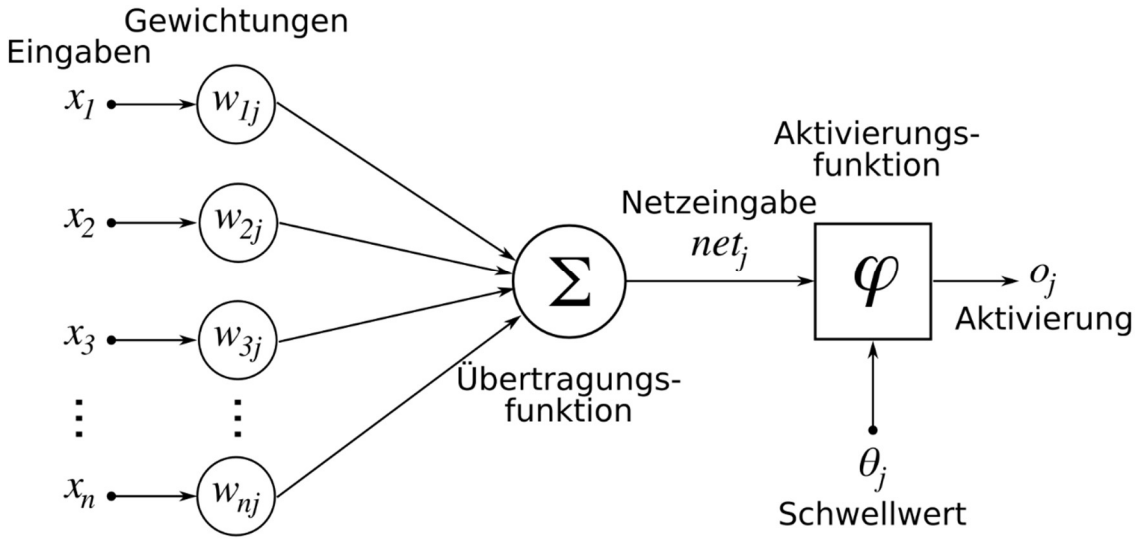


Abbildung 62: Funktionen des Machine Learnings

Die zweite eigens angefertigte Grafik veranschaulicht wie *Layers Densely Connected* sind.

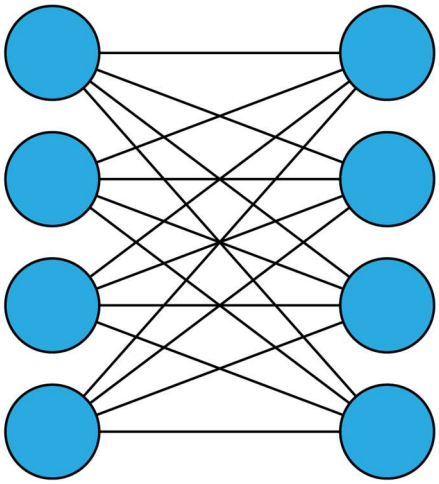


Abbildung 63: Zwei Layers Densely Connected

Die Dritte Grafik zeigt den Graphen einer *Relu Aktivierungsfunktion*.

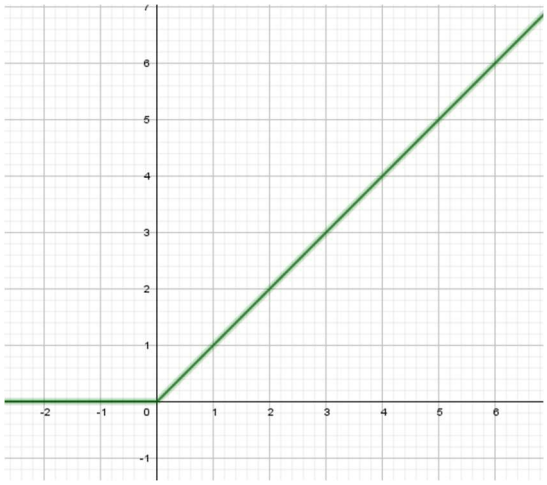


Abbildung 64: Eine Relu Aktivierungsfunktion

Mit diesen Bestandteilen ist das *Model* schon lernbereit, jedoch gibt es noch nichts zu lernen. Hier kommen die *Loss Function* und der *Optimizer* ins Spiel. Die *Loss Function* ist eine Funktion, die während des eines Trainingsvorganges die Diskrepanz zwischen den erwarteten Ergebnissen und den tatsächlich vom *Model* erbrachten Werten berechnet. [34, pp. 12-14]

Basierend auf dem Ergebnis der *Loss Function* macht sich der *Optimizer* ans Werk und passt die *Weights* der Verbindungen der Neuronen in den *Layers* so an, dass sich der ermittelte Fehler zwischen Soll- und Ist-Ausgabe, möglichst reduziert. [34, pp. 12-14]

Bei beiden (*Loss Function* und *Optimizer*) gibt es eine Vielzahl an Optionen (z.B. *Binary Crossentropy Function* und *Poisson Function* für die *Loss Function* und *RMSProp* und *Adagrad* für die *Optimizer*), die unterschiedlich funktionieren und somit für unterschiedliche Situationen vorgesehen sind.

11.1.3 Trainingsdurchlauf

Zu Beginn eines Trainingsdurchlaufs stehen als Erstes drei Datensätze: die Trainingsdaten, die Validierungsdaten und die Testdaten. Diese sind jeweils noch einmal in die eigentlichen Daten, also die Inputdaten, und die *Targets*, also die Ergebnisse dieser Daten, aufgeteilt:

- Die Trainingsdaten dienen dazu das *Model* anzulernen.
- Die Validierungsdaten dienen dazu dem *Model* während dem Trainieren Feedback zu geben.
- Die Testdaten sind Daten, die dem *Model* bis zum allerletzten Durchlauf komplett fremd sein sollen und als finaler Test gedacht sind.

Die konkreten vorhandenen Daten bestanden aus 10000 *Events* und 100000 *Values*. Die Struktur dieser Daten wurde bereits weiter oben im Dokument beschrieben (7 - DatenDaten).

Der eigentliche Testdurchlauf beginnt nun damit, dem *Model* die Trainingsdaten zu „füttern". Diese durchlaufen die *Layers* und die Ergebnisse werden mit den erwarteten *Targets* der Trainingsdaten durch die *Loss Function* verglichen. Woraufhin der *Optimizer* die *Weights* anpasst.

Wieviel Daten pro Abgleichen mit Hilfe der *Loss Function* durchgelassen werden, bestimmt die *Batch Size* [34, pp. 43-44]. Dabei werden die Trainingsdaten in Gruppen aufgeteilt, die der *Batch Size* entsprechen. Der *Optimizer* passt die *Weights* dann erst nach dem Durchlauf einer solchen Gruppe und nicht nach jedem Testdatensatz an. Je kleiner die *Batch Size*, desto feiner können die *Weights* angepasst werden, aber desto länger wird auch die Laufzeit.

Nachdem alle *Batches* abgearbeitet sind, wird das *Model* mit Hilfe der Validierungsdaten überprüft. Die Ergebnisse dieser Testläufe werden dem Programmierer als Unterstützung beim Anpassen des Models präsentiert. Jeder vollständige Durchlauf dieser Art wird *Epoch* genannt [34, pp. 68-69]. Auch hier gilt: Je größer die Zahl der Epochs, desto genauer wird das *Model*, aber desto mehr Arbeit hat auch der PC.

Erst wenn sicher ist, dass das *Model* ausreichend präzise Ergebnisse liefert, kommen die Testdaten zum Einsatz und es werden Testläufe damit durchgeführt.

11.1.4 Probleme

Während dem Trainingsdurchlauf können Probleme auftreten, die man beachten muss: Ein Problem, dass eigentlich immer auftreten wird, ist dass des sogenannten *Overfitting*. *Overfitting* bedeutet, dass das *Model* sich zu sehr darauf spezialisiert hat, die Trainingsdaten korrekt vorherzusagen und dadurch mit fremden Daten zu ungenau ist [34, pp. 94-95].

Es ist ganz normal, dass *Overfitting* ab einer gewissen *Epoch*-Zahl auftritt. Man erkennt diese Zahl daran, dass ab dann alle weiteren *Epochs* eine Erhöhung der Genauigkeit bei den Trainingsdaten, nicht aber bei den Validierungsdaten aufweist. *Overfitting* kann aber auch durch eine zu hohe Zahl an *Input/Output Units* hervorgerufen, werden. Der Grund, warum man die Testdaten nur ganz zum Schluss anwenden soll, ist, dass man verhindert, dass durch den Entwickelnden eine unabsichtliche Anpassung an diese geschieht, also *Overfitting*. Man nennt dieses Phänomen *Information Leak* [34, pp. 123-124].

Das Gegenteil des *Overfitting*, ist das *Underfitting*, das aber seltener ist. Beim *Underfitting* liefert das *Model*, wie man vielleicht schon erwartet, zu ungenaue Resultate. Ursachen dafür sind zumeist zu wenige *Input/Output Units* oder zu wenig beziehungsweise schlechte Trainingsdaten.

11.2 Die Umsetzung

11.2.1 Die Datenaufbereitung

Um nun ein solches *Model* zu erstellen, waren verschiedene Schritte notwendig. Die Daten, wie wir sie zu Testzwecken erhielten, waren zahlenmäßig ausreichend, um damit *Deep Learning* zu betreiben, jedoch waren sie noch in der Form wie sie in der firmeneigenen *MongoDB* gespeichert wurden. Das bedeutet, dass die nötigen Daten also noch von einer JSON-Form in einer zum *Machine Learning* mit *Keras* fähigen Daten-Form, nämlich *Python* Lists, konvertiert werden mussten.

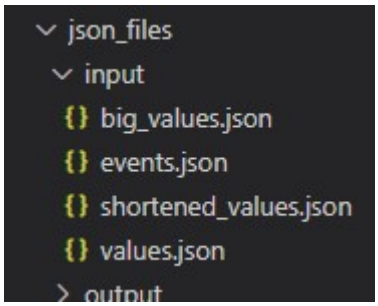


Abbildung 65: Die verschiedenen Input-Dateien

Alle dafür nötigen Funktionen befinden sich in der `DataHelper.py`, wobei die beiden Funktionen `loadFutureValues` und `loadFutureEvents` das Herzstück dieser Datei darstellen. Beide erfüllen eine ähnliche Funktion:

```
def loadFutureValues(dataFilename, targetFilename, timespan=1, valueFilter = lambda value: True, propertyFilter = lambda property: True):
```

Abbildung 66: parameter der loadFutureValues

Der `dataFilename` und der `targetFilename` geben jeweils den Namen der Dateien an, in denen jeweils die Input- und -Targetdaten für das spätere Trainieren gespeichert werden. `timespan` bestimmt, wie der Name vermuten lässt, die Zeitspanne, die zwischen zwei aufeinanderfolgenden Werten vergehen soll. Zusätzlich gibt es noch `valueFilter` und `propertyFilter`, die das Übergeben von Filterfunktionen zum Filtern von Werten direkt beim Einlesen ermöglicht. Es werden also solche Filter beim Einlesen übergeben, die beispielsweise nur die Werte eines Sensors einlesen lassen (`propertyFilte`, zu dem es auf der nächsten Seite ein Beispiel gibt). Der `valueFilter` war nur für den Fall enthalten, dass man nur bestimmte

Werte (z.B. keine negativen, nur Werte größer als zehn etc.) einlesen lassen kann, was sich aber als nicht nötig herausstellte.

```python
def loadFutureValues(dataFilename, targetFilename, timespan=1, valueFilter = lambda value: True, propertyFilter = lambda prope

    fullDataFilename = outputDir + dataFilename + fileExtension                 # concatenating the full file name
    fullTargetFilename = outputDir + targetFilename + fileExtension             # (path) of the files

    (data, targets) = checkFiles(fullDataFilename, fullTargetFilename)          # checks the files
    if not (len(data) == 0 and len(targets) == 0):                              # returns them if they are present
        print('Found output files!')
        return (data, targets)

    print('Reading the original data and generating the targets...')
    original = getValues(valueFilter = valueFilter, propertyFilter=propertyFilter)
    targets = getFutureValuesTargets(original, timespan)
    original = removeGivenIndex(original)
    print('Finished!')

    print('Normalizing data')
    normalized = normalizeValues(original)                                      # normalizes the values
    print('Length of normalized Data: ' + str(len(normalized)))
    print('Finished!')

    print('Generating JSON-string for output files...')
    targetsJson = json.dumps(targets, indent=4)                                 # converts the targets and
    normalizedJson = json.dumps(original, indent=4)                             # data into a json format string
    print('Finished!')

    writeFile(fullDataFilename, normalizedJson)                                 # writes the output in order
    writeFile(fullTargetFilename, targetsJson)                                  # to save computation time the next time

    return (original, targets)
```

Abbildung 67: Die loadFutureValues Funktion

Zuallererst werden die beiden Output-Dateinamen aus den Parametern und den fix angegebenen Variablen (`outputDir` und `fileExtension`) zusammengebaut und überprüft. Sollten bereits Dateien vorhanden sein, werden stattdessen diese zurückgegeben.

Danach wird `getValues` aufgerufen, der auch der `valueFilter` und der `propertyFilter` übergeben wird. Diese Funktion startet nun das eigentliche Einlesen (es wird `readFile` aufgerufen) und iteriert über die Ergebnisse

Von jedem Value-JSON-Objekt wird das `timestamp` Attribut zur späteren Sortierung gespeichert und das `data` JSON-Array mit Hilfe der Funktion `dataToValueList` zu einer Liste an Werten der einzelnen Sensoren umgewandelt. Hierbei wird der `propertyFilter` angewandt, wodurch nur die Messdaten bestimmter Sensoren aussortiert werden. Ein `propertyFilter` der folgendermaßen definiert ist, würde alle Werte aussortieren, die nicht vom Sensor „coc/acfc__tt402_ai“ stammen:

```
lambda propertyName: not [‚coc/acfc__tt402_ai‘].count(propertyName)==0
```

An die erste Stelle der erhaltenen Liste an Werten wird nun der zuvor gespeicherte `timestamp` drangehängt und die Listen danach sortiert. Diese 2D-Liste ist das Resultat und wird an `getValues` zurückgegeben.

```python
def getValues(valueFilter = lambda value: True, propertyFilter = lambda property: True):
    result = []
    for valueObject in readFile(inputDir + valuesFile + fileExtension, valueFilter):

        timestamp = valueObject['timestamp']                                    # gets the timestamp of the current json object
        valuesList = dataToValueList(valueObject['data'], propertyFilter)       # retrieves a list of values (using dataToValueList())...

        if (len(valuesList)) == 0:                                              # skips this iteration of the for loop if there is no data
            continue

        valuesList.insert(0, timestamp)                                         # and adds a timestamp for sorting and target generation
                                                                                # at the first index
        result.append(valuesList)

    result.sort(key=lambda list: list[0])                                       # sorting by the first index of every list (timestamp)

    return result
```

Abbildung 68: Die getValues-Funktion

Damit sind nun alle Werte, so wie sie aus den *JSON* Dateien ausgelesen werden konnten, eingelesen. Es folgt der Aufruf von `getFutureValuesTargets`, die als Parameter die bereits eingelesenen Wertelisten und die `timespan` übergeben bekommt. Diese Funktion ermittelt nun unter Zuhilfenahme der Wertelisten und zusätzlichen Helper-Funktionen (wie `getNextValue`, die den um `timespan` verschobenen Wert zu einem anderen Wert sucht) für jeden Wert den zugehörigen zukünftigen Wert. Für die *Events* gibt es eine ähnliche Funktion, die später in diesem Unterpunkt erklärt wird.

```python
def getFutureValuesTargets(lists, timespan):                # returns all targets for the training data if future values are needed.
    targets = []                                            # Timespan is the amount of seconds until the future value of a given value

    for singleList in lists:
        target = getNextValue(lists, singleList[0], timespan)  # retrieves the next value given the settings (timespan) and the inputs

        if not target == -1:                                # if no value was found getNextValue() returns -1
            targets.append(target)
    return targets
```

Abbildung 69: Die getFutureValuesTargets-Funktion

Ist nun auch das erledigt, werden aus den ursprünglichen Wertelisten der `timestamp`-Wert entfernt, da dieser nur dazu da war, einen korrekten chronologischen Verlauf sicherzustellen, aber nicht Teil der Trainingsdaten ist. Das geschieht mit Hilfe der Funktion `removeGivenIndex`.

Den Ablauf der `loadFutureValues` folgend, die bereits oben teilweise erklärt wurde, müssen noch die einzelnen Messwerte der Sensoren normalisiert werden. Normalisieren bedeutet hier, die Werte von einem konkreten und zu großen Wertebereich, zu Werten in einem Wertebereich zwischen null und eins zu konvertieren. Dieser Vorgang wird auch *Feature Scaling* genannt [36].

Es gibt verschiedene Herangehensweisen, um das zu erreichen. Die Entscheidung fiel auf Min-Max-Normalization, die das, wie der Name verrät, Minimum und Maximum der Werte (also die Grenzen des Wertebereichs) zur Berechnung heranzieht. Die Formel für die Berechnung eines normalisierten Wertes sieht also so aus (bezogen von [36]):

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Bei der Implementierung dieser Normalisierung mussten natürlich noch ein paar Sonderfälle beachtet werden. Manche Sensoren waren noch nicht in Betrieb und lieferten ständig den Wert Null, bei anderen waren die Werte nur im Bereich null bis eins enthalten und waren damit eigentlich schon zum Weiterverwenden bereit. Im ersteren Fall käme es zu einer Division durch null, da die Differenz zwischen dem Maximum, das null ist, und dem Minimum, das auch null ist, null ist.

```python
def normalizeValues(lists):                                 # normalizes all given values so that
                                                            # they are within the range of 0-1
                                                            # scaled off the minimum 0 and the maximum 1
    npLists = np.array(lists)

    min = np.min(npLists, axis=0)                           # gets the minimum of one row (axis=0)
    max = np.max(npLists, axis=0)                           # the same but for maximum

    #npLists -= npLists.mean(axis=0)
    #npLists /= npLists.std(axis=0)                         # this will produce NaNs because of a
                                                            # division by zero and unfortunately cannot be used
    result = np.zeros(npLists.shape)

    for i, list in enumerate(npLists):                      # iterates over all lists
        for i2, value in enumerate(list):                   # iterates over all entries in the current list
            maxMinDiff = max[i2] - min[i2]                  # computes the difference between max and min

            if (value == 1 or value == 0):                  # filters for 1 and 0 because these values
                result[i][i2] = value                       #  do not need to be normalized

            if (maxMinDiff != 0):                           # filters for 0 to avoid division by zero
                result[i][i2] = (value - min[i2]) / maxMinDiff     # calculate the normalized value and add it to result

    return result
```

Abbildung 70: Normalisieren der Werte mit der normalizeValues-Funktion

Nachdem nun auch dieser Schritt getan wurde, müssen aus den Listen nur noch JSON-Strings erstellt werden was mit `json.dumps` möglich war. Zu guter Letzt muss dieser String noch in die zuvor angegebenen Output-Dateien geschrieben werden. Das geschieht mit der eigens dafür erstellten Helper-Funktion `writeFile`.

Wie bereits eingangs erwähnt gibt es zusätzlich zur `loadFutureValues` auch `loadFutureEvents`. Diese Funktion ist grundsätzlich sehr ähnlich in ihrer Umsetzung, jedoch gibt es einen Unterschied bei der Suche der zukünftigen Werte. Statt nach dem nächsten *Event* nach einer gewissen Zeitspanne zu suchen, wird nach dem frühesten Vorkommnis des gleichen *Events* gesucht. Was als gleich erachtet wird, kann mit Hilfe des `eventFilter`-Parameters der Funktion bestimmt werden. Soll also das zukünftige *Event* zu einem *Event* mit `severity` zwei gesucht werden kann man das beispielsweise mit folgendem `eventFilter` erreichen:

```
eventFilter = lambda event: event['data']['severity'] == 2
```

11.2.2 Trainingsdaten Helper-Funktionen – ModelUtil

Beim eigentlichen Trainieren des *Models* gab es immer wieder Prozeduren, die sich wiederholten oder sich generell dazu anboten, in eine eigene *Python*-Datei ausgelagert zu werden: Aus diesem Grund entstand die `ModelUtil.py`-Datei und Klasse.

Sie hat drei „Objektvariablen“: `epochs`, `batch_size` und `k`. `epochs` und `batch_size` entsprechen den jeweiligen Stellschrauben beim *Machine Learning* mit *Keras*. `k` dient der *K-Fold-Validation*, die an entsprechender Stelle erläutert wird (Siehe Punkt 11.2.2.3).

11.2.2.1 split_data

Die `split_data`-Funktion nimmt `data`, `targets` und `split_at` entgegen. `data` entspricht der Gesamtmenge an Trainingsdaten, die nicht *Targets* sind. Dementsprechend sind `targets` die dazugehörigen *Targets*. Mit Hilfe von `split_at` wird angegeben, an welcher Stelle jeweils `data` und `targets` geteilt werden sollen.

```
def split_data(self, data, targets, split_at):
    (training_data, training_targets) = (data[:split_at], targets[:split_at])
    (test_data, test_targets) = (data[split_at:], targets[split_at:])

    np_training_data = np.array(training_data).astype('float32')
    np_training_targets = np.array(training_targets).astype('float32')

    np_test_data = np.array(test_data).astype('float32')
    np_test_targets = np.array(test_targets).astype('float32')

    return ((np_training_data, np_training_targets), (np_test_data, np_test_targets))
```

Abbildung 71: Die split_data-Funktion

Das Resultat dieser Funktion ist ein Tupel aus Tupeln aus *NumPy*-Arrays, mit denen *Keras* umgehen kann. Ein Tupel ist eine simple Datenstruktur in *Python*, die aus zwei Werten besteht und *NumPy* ist eine Bibliothek für mathematische Operationen in *Python* [37].

11.2.2.2 test_model

Die `test_model`-Funktion nimmt die nötigen Trainings- und Testdaten (`training_data`, `training_targets`, `test_data`, `test_targets`) entgegen und nutzt diese gemeinsam mit den zuvor konfigurierten Variablen, die die *Batch Size* und *Epochs* bestimmen.

```
def test_model(self, training_data, training_targets, test_data, test_targets, model):
    model.fit(training_data, training_targets, batch_size = self.batch_size, epochs = self.epochs)
    history = model.evaluate(test_data, test_targets, batch_size=self.batch_size)
    return history
```

Abbildung 72: Die test_model-Funktion

Es wird eine `History` beim Evaluieren erzeugt, die Informationen über den Erfolg beinhaltet, welche zum Schluss zurückgegeben wird.

11.2.2.3 train

Das Trainieren in der Funktion `train` basiert auf dem Prinzip der *K-Fold-Validation*. Dabei werden die Gesamtmenge an verfügbaren Trainingsdaten und *Targets* auf k Teile aufgeteilt und k-*Models* instanziiert. Jedes *Model* nutzt nun einen Teil der Trainingsdaten und *Targets* zur Validierung, während der Rest zum Trainieren genutzt wird. Der Teil der jedoch, der der Validierung dient, ist bei jedem *Model* ein anderer. Zum Schluss wird der Durchschnitt aller gewünschten Trainingsmetriken (z. B. *Mean Absolute Error*, der weiter unten erklärt wird) gebildet. Diese Trainingsfunktion wurde zum Erproben der Konfiguration der *Models* verwendet, jedoch nicht für die finalen *Models*. Das rührt daher, dass in einer früheren Phase des Trainierens fälschlicherweise davon ausgegangen wurde, dass die bereitgestellten Daten in Form von *JSON* Dateien zu wenig sind.

Abbildung 73: Wie K-Fold-Validation funktioniert (schematisch)

11.2.3 Das eigentliche Trainieren – ModelLearning

Die `ModelLearning.py` ist die Datei, in der alle vorher erklärten Dateien verwendet werden. Zu Beginn werden, um das Anpassen des Trainings zu erleichtern, alle in der *ModelUtil* und davor erklärten Metriken (`batch_size`, `epochs`, k und `split_at`) definiert. Zusätzlich dazu gibt es eine boolesche Variable `isTraining`, die bestimmt, ob beim Ausführen der Datei trainiert oder getestet werden soll.

```
# --- training configuration ----

k = 4                   # this training uses k-fold validation (eg. 4-fold, 3-fold, ...)
epochs = 20             # number of training loops per model
batch = 1
split_at = 8000         # the index at which the input data is split
isTraining = True       # wether to train or to test
```

Abbildung 74: Trainingskonfiguration eines Models

Der nächste wichtige Bestandteil ist die Funktion `buildModel`. Diese Funktion instanziiert einfach ein fix in der Funktion vorgegebenes *Model* und ermöglicht damit eine einfache Anpassung desselbigen, da sie überall dort verwendet wird, wo ein *Model* gebraucht wird. Alle hier und später dargestellten Konfigurationen der *Models* wurden durch Erproben ermittelt, vor allem die Anzahl der *Input/Output Units und Layers.*

```
def buildModel():
    model = models.Sequential()

    model.add(layers.Dense(200, activation='relu', input_shape=(np_training_data.shape[1],)))
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model
```

Abbildung 75: Die buildModel-Funktion

Der erste wirkliche Code, der in der `ModelLearning` aufgerufen wird, ist dieser:

```
(data, targets) = dh.loadFutureEvents('events_2_data', 'events_2_targets')    # loads the whole data and the targets
print('Data: ' + str(len(data)))
print('Targets: ' + str(len(targets)))

modelUtil = ModelUtil(k, epochs, batch)
(np_training_data, np_training_targets), (np_validation_data, np_validation_targets) = modelUtil.split_data(data, targets, split_at)

print('Loading and preparing data was successful!\nTraining configuration:\n')

print('- k = ' + str(k))
print('- epochs = ' + str(epochs))
print('- batch size = ' + str(batch))
```

Abbildung 76: Laden und bearbeiten der Daten

Es werden mit Hilfe des `DataHelper` (importiert als dh) die nötigen Datensets geladen und simple Informationen (die Länge der Lists) zu diesen ausgegeben, um sofort überprüfen zu können, ob alles einwandfrei eingelesen wurde. Danach wird ein Objekt der `ModelUtil`-Klasse angelegt und mit den entsprechenden Werten initialisiert, dass auch direkt zum Aufteilen der Daten genutzt wird.

Nach diesem ersten Block folgt eine simple Abfrage nach `isTraining`. Ist diese erfolgreich, werden alle nötigen Schritte eingeleitet, um ein Model zu trainieren. Dazu wird zunächst einmal

mit Hilfe der `buildModel` ein *Model* erzeugt, dass dann auf die vorher eingelesenen und bereinigten Daten, mit der angegebenen *Batch Size* und Zahl an *Epochs*, trainiert wird.

Der Rückgabewert dieses Trainings ist eine Historie, die alle beim Instanziieren des *Models* (`metrics` Parameter der Funktion `compile`) genannten Metriken mitprotokolliert. Diese werden dann mit Hilfe von *Matplotlib* [38] dargestellt, um leichter Anpassungsentscheidungen treffen zu können. Auf die Bedeutung dieser Grafik wird an späterer Stelle eingegangen (Siehe Punkt 11.2.5.1).

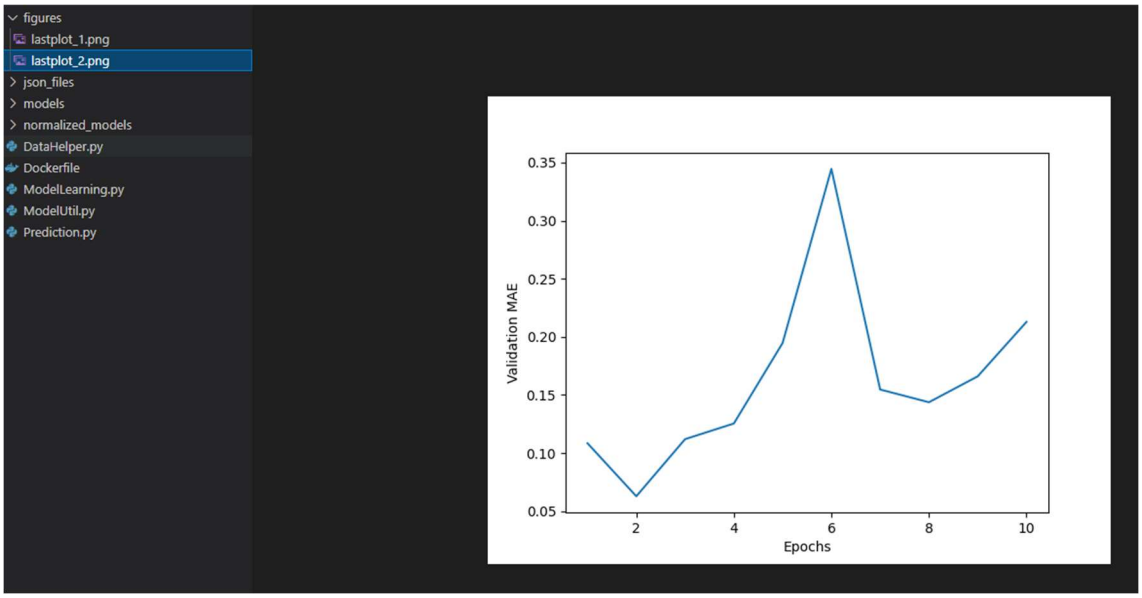


Abbildung 77: Darstellung der Historie-Daten mit Matplotlib

Es folgt eine Abfrage, um zu bestimmen, ob das trainierte *Model* verworfen oder gespeichert werden soll. Im Falle des Speicherns kann noch ein Name angegeben werden, der dann an die `save`-Funktion des *Keras Models* übergeben wird. Daraufhin speichert *Keras* alle nötigen Dateien in einem Ordner mit dem angegebenen Namen, der genutzt werden kann, um mit `load_model` dieses *Model* wieder zu laden und verwenden.

Sollte das Model stattdessen getestet werden, werden die Daten, die eigentlich der Validierung dienen würden, zum Testen verwendet. Nach dem Testen wird die Historie in diesem Fall auf der Konsole ausgegeben.

11.2.4 Verwendung im Backend – Prediction

Abschließend soll es nun für das Backend möglich sein, diese *Models* zu nutzen, um Vorhersagen an das Frontend zu schicken. Dazu wird vom Backend (wie bereits beschrieben in Punkt 9.3.2.3)

eine *Python*-Datei, nämlich die `Prediction.py`, mit entsprechenden Kommandozeilenparametern aufgerufen. Zusätzlich dazu müssen auch noch die entsprechenden, also für das Frontend notwendigen, *Models* manuell auf das System des Backends gebracht werden.

Der erste Kommandozeilenparameter, `identifier`, bestimmt, welches *Model* aufgerufen wird. Der `identifier` kann dabei der Name eines Sensors oder Events sein. Außerdem gibt es noch die Variable `model_name`, die im Zuge der Überprüfung der Kommandozeilenparameter so gewählt werden muss, dass am Ende das richtige *Model* geladen wird.

Sollte der `identifier` „event“ sein, wird der zweite Kommandozeilenparameter eingelesen, der die `severity` repräsentiert. Ist die eingelesene `severity` ein Wert von eins bis sechs, wird der dritte Kommandozeilenparameter eingelesen, der die Werte aller Sensoren zu einem gewissen Zeitpunkt in JSON-Form beinhaltet (es ist das `data`-Objekt aus den `values`). Stimmt seine Länge mit der Anzahl der Sensoren überein, die in `number_of_sensors` gespeichert wird, gilt der Aufruf als gültig und `model_name` wird entsprechend angepasst. Das bedeutet, dass der Name des Sensors, dessen Wert vorherzusagen ist, mit den dafür nötigen Zeichen zu einem Pfad kombiniert wird, an dem sich das benötigte *Model* befindet.

Sollte der `identifier` jedoch ein anderer sein, wird überprüft, ob der zweite Kommandozeilenparameter vom Typ *Float* ist. Das geschieht, da der zweite Parameter der derzeitige Wert des gewünschten Sensors sein muss, wenn der Wert eines Sensors vorhergesagt werden soll. Ist auch das erfolgreich, gilt der Aufruf ebenso als gültig und auch `model_name` wird entsprechend angepasst.

Nun da alle nötigen Parameter erkannt und der `model_name` passen sollten, wird überprüft, ob `model_name` noch immer, wie am Anfang initialisiert, ein leerer *String* ist, da ansonsten nichts passiert. Ist das nicht der Fall werden die übergebenen Daten in eine für das *Model* verwendbare Form konvertiert, das *Model* geladen, der Wert vorhergesagt und dann in eine Output-Datei gespeichert. Diese wird dann vom Backend ausgelesen.

11.2.5 Verschiedene Trainingsdurchläufe

Im Folgenden sind ein paar *Model*-Konfigurationen, deren finaler *MAE* (*Mean Absolute Error/mae* = dt. Durchschnittlicher absoluter Fehler) auf den Trainingsdaten und der *MAE* der Validierungsdaten (*val_mae*). Der *MAE* berechnet sich aus der Summe aller Differenzen zwischen den prognostizierten und den tatsächlichen Wert, die dann durch die Zahl aller Werte geteilt wird [39]. Je niedriger der MAE, desto besser.

11.2.5.1 Sensor acfc_tt402_ai

Das *Model* für diesen Sensor war folgendermaßen konfiguriert.

```
# --- training configuration ----

k = 4                # this training uses k-fold validation (eg. 4-fold, 3-fold, ...)
epochs = 20           # number of training loops per model
batch = 1
split_at = 700       # the index at which the input data is split
isTraining = True    # wether to train or to test

def buildModel():
    model = models.Sequential()

    model.add(layers.Dense(4, activation='relu', kernel_regularizer=regularizers.l2(0.022), input_shape=(np_training_data.shape[1],)))
    model.add(layers.Dense(4, activation='relu', kernel_regularizer=regularizers.l2(0.023)))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model
```

Abbildung 78: Konfiguration für Sensor "acfc_tt402_ai"

Diese Konfiguration führte zu folgenden Ergebnissen. Am Graph kann man erkennen, dass die Genauigkeit in der Vorhersage zunimmt (und der *VAE* abnimmt), je mehr *Epochs* durchlaufen wurden. Diese Entwicklung kann man oft beobachten. Es gibt jedoch auch Ausnahmen, bei denen eine niedrigere *Epoch* Zahl vorteilhaft sein kann (wie bei 11.2.3 am Ende dargestellt).

```
Epoch 20/20
700/700 [==============================] - 2s 3ms/step - loss: 9.9038e-04 - mae: 0.0234 - val_loss: 0.0076 - val_mae: 0.0846

Saving MAE values as graph...
Finished!
```

Abbildung 79: Metriken für Sensor "acfc_tt402_ai"

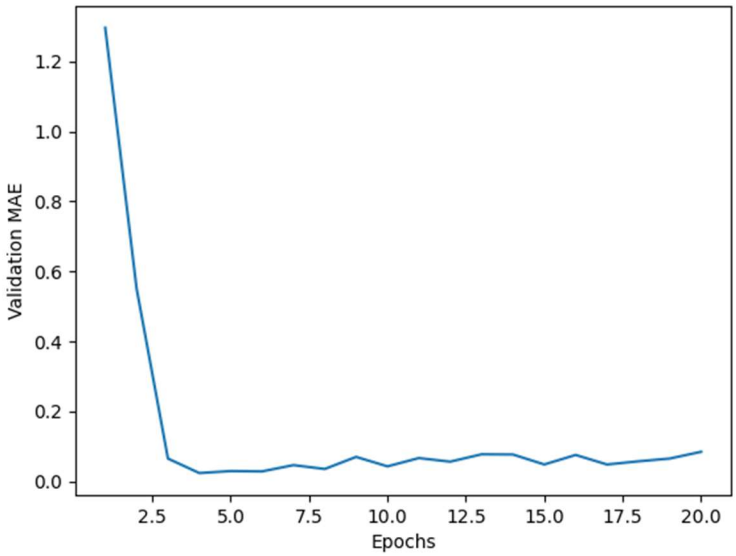


Abbildung 80: Graph zu den Metriken von Sensor "acfc_tt402_ai"

11.2.5.2 Sensor cel_tt103_ai

Das *Model* für diesen Sensor war folgendermaßen konfiguriert.

```
# --- training configuration ----

k = 4                 # this training uses k-fold validation (eg. 4-fold, 3-fold, ...)
epochs = 20           # number of training loops per model
batch = 1
split_at = 700        # the index at which the input data is split
isTraining = True     # wether to train or to test

def buildModel():
    model = models.Sequential()

    model.add(layers.Dense(4, activation='relu', kernel_regularizer=regularizers.l2(0.02325), input_shape=(np_training_data.shape[1],)))
    model.add(layers.Dense(4, activation='relu', kernel_regularizer=regularizers.l2(0.024)))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model
```

Abbildung 81: Konfiguration für Sensor "cel_tt103_ai"

Diese Konfiguration führte zu folgenden Ergebnissen.

```
Epoch 20/20
700/700 [==============================] - 2s 3ms/step - loss: 5.2284 - mae: 0.5178 - val_loss: 0.1885 - val_mae: 0.2051

Saving MAE values as graph...
Finished!
```

Abbildung 82: Metriken für Sensor "cel_tt103_ai"

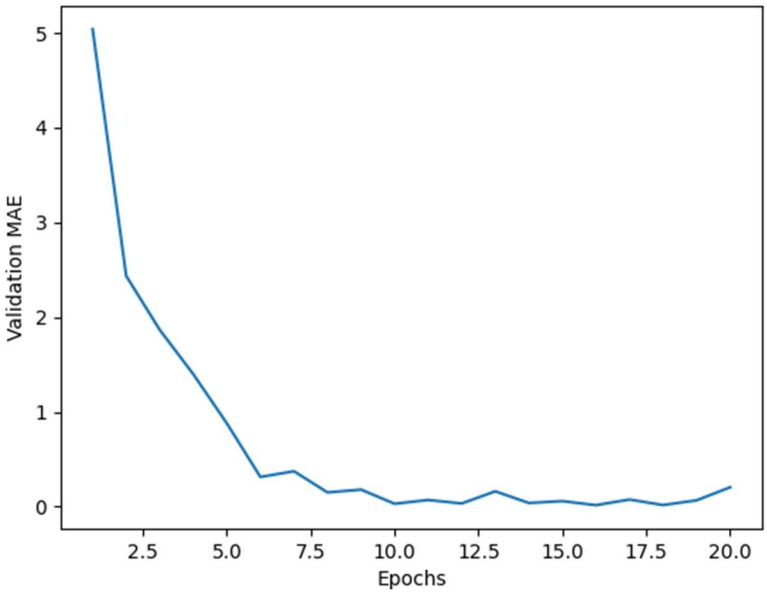


Abbildung 83: Graph zu den Metriken von Sensor "cel_tt103_ai"

11.2.5.3 Sensor cel_tt104_ai

Das *Model* für diesen Sensor war folgendermaßen konfiguriert.

```
# --- training configuration ----

k = 4                 # this training uses k-fold validation (eg. 4-fold, 3-fold, ...)
epochs = 20           # number of training loops per model
batch = 1
split_at = 700        # the index at which the input data is split
isTraining = True     # wether to train or to test

def buildModel():
    model = models.Sequential()

    model.add(layers.Dense(4, activation='relu', kernel_regularizer=regularizers.l2(0.026), input_shape=(np_training_data.shape[1],)))
    model.add(layers.Dense(4, activation='relu', kernel_regularizer=regularizers.l2(0.025)))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    return model
```

Abbildung 84: Konfiguration für Sensor "cel_tt104_ai"

Diese Konfiguration führte zu folgenden Ergebnissen.

```
Epoch 20/20
700/700 [==============================] - 2s 3ms/step - loss: 4.5967 - mae: 0.5063 - val_loss: 0.1117 - val_mae: 0.0623

Saving MAE values as graph...
Finished!
```

Abbildung 85: Metriken für Sensor "cel_tt104_ai"

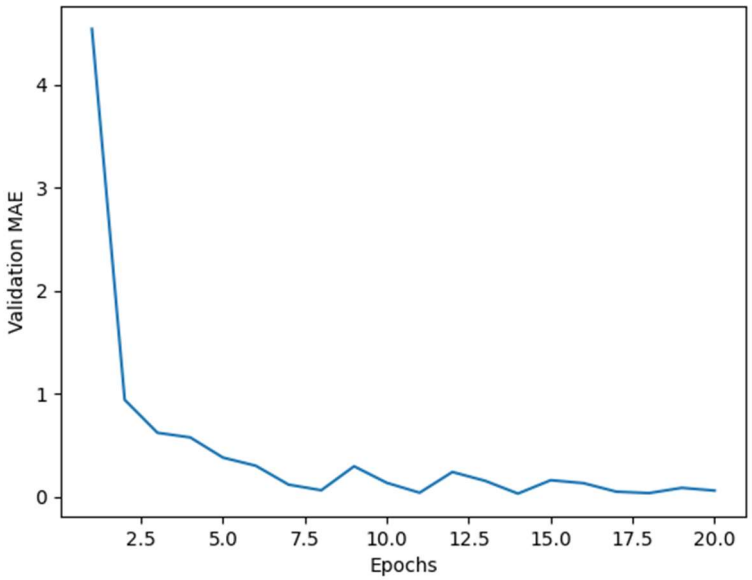


Abbildung 86: Graph zu den Metriken von Sensor "cel_tt104_ai"

11.2.5.4 Ein Event Model

Es gab jedoch auch *Models*, deren Training nicht optimal verlaufen ist. Dieses Training spiegelt sich auch in dem Graphen wider.

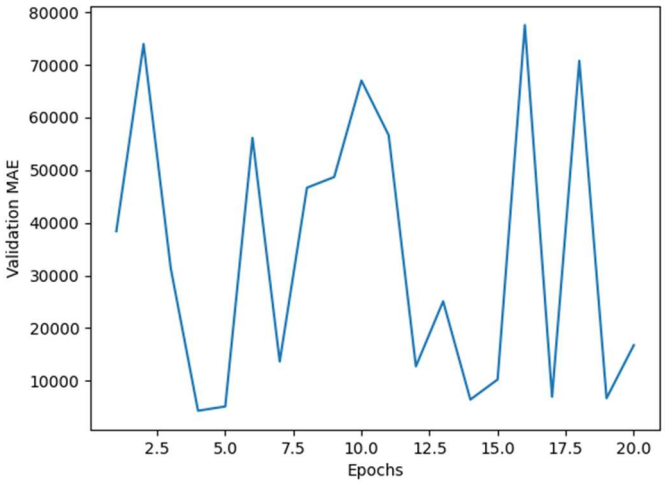


Abbildung 87: Graph zu den Metriken eines Event Models

Quellen und Literaturverzeichnis

[1] J. Lau, „Der Weg zum grünen Wasserstoff - Forschung Spezial - derStandard.at › Wissenschaft,“ 25 August 2021. [Online]. Available: https://www.derstandard.at/story/2000129133245/der-weg-zum-gruenen-wasserstoff. [Zugriff am 18 März 2022].

[2] F. I. GmbH, „Fronius Solhub - Green hydrogen with solar energy,“ [Online]. Available: https://www.fronius.com/en/solar-energy/installers-partners/products-solutions/commercial-energy-solutions/green-hydrogen-with-solar-energy-solhub. [Zugriff am 14 03 2022].

[3] Microsoft, „Microsoft Github,“ [Online]. Available: https://microsoft.github.io/code-with-engineering-playbook/developer-experience/devcontainers/. [Zugriff am 3 1 2022].

[4] Docker, „Why Docker?,“ [Online]. Available: https://www.docker.com/why-docker. [Zugriff am 18 02 2022].

[5] Docker, „What is a Container?,“ [Online]. Available: https://www.docker.com/resources/what-container. [Zugriff am 2022 2 18].

[6] Redhat, „What is a REST API?,“ 08 05 2020. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. [Zugriff am 18 02 2022].

[7] Wikipedia, „MongoDB,“ [Online]. Available: https://de.wikipedia.org/wiki/MongoDB. [Zugriff am 27 12 2021].

[8] OpenJS Foundation, „About | Node.js,“ [Online]. Available: https://nodejs.org/en/about/. [Zugriff am 2022 03 01].

[9] OpenJS Foundation, „Express - Node.js web application framework,“ [Online]. [Zugriff am 19 02 2022].

[10] „Flutter (software) - Wikipedia,“ [Online]. Available: https://en.wikipedia.org/wiki/Flutter_(software). [Zugriff am 12 März 2022].

[11] „Flutter - Build apps for any screen,“ [Online]. Available: https://flutter.dev/?gclsrc=ds&gclsrc=ds. [Zugriff am 12 März 2022].

[12] „Dart (programming language) - Wikipedia,“ [Online]. Available: https://en.wikipedia.org/wiki/Dart_(programming_language). [Zugriff am 12 März 2022].

[13] „Dart programming language | Dart,“ [Online]. Available: https://dart.dev/. [Zugriff am 12 März 2022].

[14] C. Voskoglou, „What is the best programming language for Machine Learning? | by Developer Nation | Towards Data Science,“ 5 5 2017. [Online]. Available: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7. [Zugriff am 10 3 2022].

[15] F. Chollet, „François Chollet - Personal Page,“ [Online]. Available: https://fchollet.com/. [Zugriff am 24 3 2022].

[16] R. Thomas, „Big deep learning news: Google Tensorflow chooses Keras · fast.ai,“ 3 1 2017. [Online]. Available: https://www.fast.ai/2017/01/03/keras/. [Zugriff am 24 3 2022].

[17] MongoDB, „Aggregation - MongoDB Manual,“ [Online]. Available: https://docs.mongodb.com/manual/aggregation/. [Zugriff am 03 02 2022].

[18] MongoDB, „Aggregation Pipeline Optimization - MongoDB Manual,“ [Online]. Available: https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/. [Zugriff am 21 02 2022].

[19] MongoDB, „$addFields(aggregation) - MongoDB Manual,“ [Online]. Available: https://docs.mongodb.com/manual/reference/operator/aggregation/addFields/. [Zugriff am 03 02 2022].

[20] MongoDB, „$project(aggregation) - MongoDB Manual,“ [Online]. Available: https://docs.mongodb.com/manual/reference/operator/aggregation/project/. [Zugriff am 20 02 2022].

[21] MongoDB, „$match(aggregation) - MongoDB Manual,“ [Online]. Available: https://docs.mongodb.com/manual/reference/operator/aggregation/match. [Zugriff am 21 02 2022].

[22] MongoDB, „$limit(aggrgeation - MongoDB Manual),“ [Online]. Available: https://docs.mongodb.com/manual/reference/operator/aggregation/limit/. [Zugriff am 22 02 2022].

[23] MongoDB, „$unwind(aggregation) - MongoDB Manual,“ [Online]. Available: https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/. [Zugriff am 22 02 2022].

[24] „Navigation and routing | Flutter,“ [Online]. Available: https://docs.flutter.dev/development/ui/navigation. [Zugriff am 12 März 2022].

[25] „Flutter documentation | Flutter,“ [Online]. Available: https://docs.flutter.dev/. [Zugriff am 13 März 2022].

[26] „multi_select_flutter | Flutter Package,“ [Online]. Available: https://pub.dev/packages/multi_select_flutter. [Zugriff am 12 März 2022].

[27] „FutureBuilder class - widgets library - Dart API,“ [Online]. Available: https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html. [Zugriff am 2022 März 12].

[28] „data_table_2 | Flutter Package,“ [Online]. Available: https://pub.dev/packages/data_table_2. [Zugriff am 12 März 2022].

[29] „DataCell class - material library - Dart API,“ [Online]. Available: https://api.flutter.dev/flutter/material/DataCell-class.html. [Zugriff am 12 März 2022].

[30] „charts_flutter | Flutter Package,“ [Online]. Available: https://pub.dev/packages/charts_flutter. [Zugriff am 2022 März 12].

[31] „TimeSeriesChart class - flutter library - Dart API,“ [Online]. Available: https://pub.dev/documentation/charts_flutter/latest/flutter/TimeSeriesChart-class.html. [Zugriff am 2022 März 12].

[32] „Series class - common library - Dart API," [Online]. Available: https://pub.dev/documentation/charts_common/latest/common/Series-class.html. [Zugriff am 12 März 2022].

[33] S. Omojola, „How to create Flutter charts with charts_flutter - LogRocket Blog," [Online]. Available: https://blog.logrocket.com/how-create-flutter-charts-with-charts-flutter/. [Zugriff am 12 März 2022].

[34] F. Chollet, Deep Learning with Python, Manning Publications Co., 2018.

[35] ChrisIb, „ArtificialNeuronModel deutsch - Künstliches Neuron – Wikipedia," 14 7 2005. [Online]. Available: https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron#/media/Datei:ArtificialNeuronModel_deutsch.png. [Zugriff am 27 3 2022].

[36] B. Roy, „All about Feature Scaling. Scale data for better performance of… | by Baijayanta Roy | Towards Data Science," 6 4 2020. [Online]. Available: https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35. [Zugriff am 9 3 2022].

[37] „NumPy," [Online]. Available: https://numpy.org/about/. [Zugriff am 29 3 2022].

[38] J. D. Hunter, „Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering, vol. 9, no. 3,* pp. 90-95, 5 2007.

[39] Keras Special Interest Group, „Regression losses," [Online]. Available: https://keras.io/api/losses/regression_losses/. [Zugriff am 14 03 2022].

[40] OpenJS Foundation, „Using express middleware," [Online]. Available: https://expressjs.com/en/guide/using-middleware.html#middleware.router. [Zugriff am 21 02 2022].

[41] NumPy Steering Council, „NumPy," [Online]. Available: https://numpy.org/about/. [Zugriff am 10 3 2022].

[42] Python Software Foundation, „3.10.2 Documentation," 13 3 2022. [Online]. Available: https://docs.python.org/3/. [Zugriff am 13 3 2022].

Verzeichnis der Abbildungen, Tabellen und Abkürzungen

Abkürzung	Bedeutung
REST	Representational State Transfer
JSON	*JavaScript* Object Notation
URL	Uniform Resource Locator
API	Application Programming Interface
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
JS	*JavaScript*