Technische Hochschule Ostwestfalen-Lippe
University of Applied Sciences and Arts

Sommersemester 2022

Auswertung von Pegelständen

Modularbeit

Vorgelegt im Kontext des Moduls „Datenerfassung und Datenhaltung 2“

am Fachbereich Technische Informatik und Elektrotechnik
im Studiengang Data Science

Veranstalter:	Prof. Dr. Philipp Bruland
Vorgelegt von:	Jan Lippemeier
	Jahnstraße 21
	32805 Horn-Bad Meinberg
	0176/879017819
	jan.lippemeier@stud.th-owl.de
Matr. Nr.:	15466003
Vorgelegt von:	Henrik Lohre
	Kastanienweg 28
	32839 Steinheim
	05233/93919
	henrik.lohre@stud.th-owl.de
Matr. Nr.:	15465094
Vorgelegt von:	Dennis Reinhardt
	Heustraße 45
	32657 Lemgo
	0157/34989875
	dennis.reinhardt@stud.th-owl.de
Matr. Nr.:	15465006
Abgabetermin:	22.08.2022

22. August 2022

Inhaltsverzeichnis

Abkürzungen

API	Application Programming Interface
ASP.NET	Active Server Pages.NET
CORS	Cross Origin Resource Sharing
D.R.	Dennis Reinhardt
DBMS	Datenbank-Management System
DML	Data Manipulation Language
DQL	Data Query Language
EF	Entity Framework
ER	Entity Relation
H.L.	Henrik Lohre
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDNR	Identification Number
IP	Internet Protocol
ISO	International Organization for Standardization
J.L.	Jan Lippemeier
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
REST-API	Representational State Transfer - Application Programming Interface
SQL	Structured Query Language
SSL	Secure Sockets Layer
UTC	Universal Time Coordinated
VPS	Virtual Private Server

1 Einleitung (D.R.)

Im Rahmen des Moduls *Datenerfassung und Datenhaltung 2* wurde ein Projekt mit dem Thema Pegelstände erstellt. Dieses stellt über eine Webapplikation die aktuellen und historischen Daten der Flüsse und Seen innerhalb Deutschlands dar. Die Themenwahl begründet sich durch die Aktualität des Klimawandels und den daraus folgenden Gefahren und Risiken. Es entstehen häufiger Wetterextreme und damit treten sowohl Überschwemmungen als auch Dürren auf. Mögliche Folgen sind:

1. Wassernotstände
2. Wirtschaftliche Schäden
3. Gefährdung von Gesundheit und Leben
4. Brandrisiken
5. Artensterben

Mit diesem Projekt sollen die oben genannten Folgen verringert werden, indem die über eine Website (Pegelonline.de) bezogenen Wasserdaten gesichert und diese in einer Webapplikation bereitgestellt werden. Die Datenhaltung bietet sowohl die Möglichkeit einer Risikoanalyse durch eine Auswertung als auch die Möglichkeit einer Vorhersage unter Verwendung verschiedener Lernalgorithmen. Die Applikation kann diese Daten anschließend grafisch darstellen und damit der Öffentlichkeit zur Verfügung stellen.

2 Konzept (H.L.)

Durch unsere Teamzusammensetzung und unser Vorwissen haben wir uns für diesen Projektaufbau entschieden. Dies war vorteilhaft, da wir parallel anfangen konnten zu arbeiten, ohne auf die anderen warten zu müssen. Die beruflichen Kenntnisse und Aufgaben haben uns in vielen Bereichen geschult und vorbereitet. Das Vorwissen was wir zusammen einbrachten ist weit gestreut in den Bereichen C#, Webentwicklung, APIs oder Docker. Diese Projektarbeit hat uns die Chance geboten viel von diesen Fähigkeiten abzurufen und anzuwenden. Unsere Methoden und der Umgang mit dem Material der Daten bot großes Potenzial erneut an diesen Fähigkeiten zu feilen. Zusätzlich lernten wir viele neue Wege und Methoden, um so eine Aufgabe zu bewältigen. So entstand die Idee für diese Projektskizze.

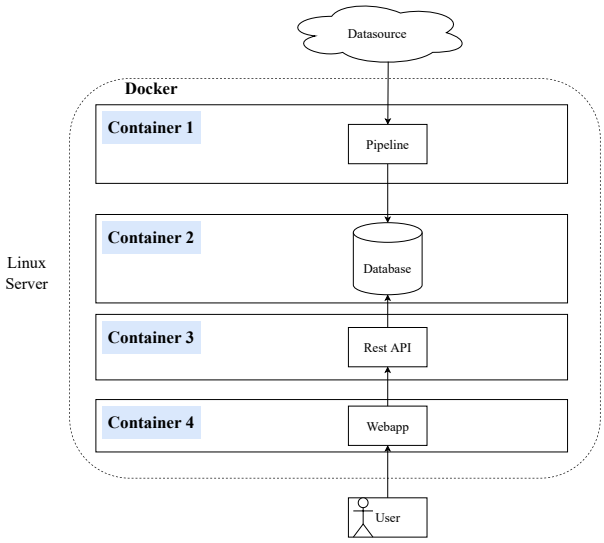

Abbildung 1: Skizze unseres Konzeptes

3 Nginx (H.L.)

Nginx [19] ist ein Open-Source-Webserver, der mehrere Funktionen einnehmen kann, zum Beispiel als Reverse-Proxy, HTTP-Cache oder Load-Balancer. Vorteile von Nginx sind, dass dieser Webserver eine geringe Speichernutzung besitzt und bei Webanforderungen eine hohe Parallelität bietet. Der Server basiert auf einem asynchronen, ereignisgesteuerten Ansatz. Anforderungen werden in einem einzigen Thread verarbeitet. Es gibt einen Master-Prozess der mehrere Worker-Prozesse steuert, dies erlaubt einen hohen Datendurchsatz. Ngnix wurde in diesem Projekt sowohl für das Frontend als auch für den Server (Subdomain und HTTPS) verwendet (siehe Kapitel 4).

4 Server (J.L.)

Der Server dient als Plattform für Entwicklungszwecke und zur Publikation der Weboberfläche für Externe. Dazu wird dies auf einem *Virtual Private Server*(VPS) gehostet. Der Server verfügt über das Betriebssystem Ubuntu 20.04.4 LTS [24]. Um die Verfügbarkeit der Weboberfläche gewährleisten zu können, wird wie in Kapitel 3 beschrieben Nginx verwendet.

4.1 Subdomain

Für das Projekt wurde die Subdomain dd2.janlippemeier.de mittels des folgenden Ausschnitts aus der Konfigurationsdatei von Nginx verwendt:

```
server{
 if ($host = dd2.janlippemeier.de) {
  return 301 https://$host$request_uri;
 }
 listen 80;
 server_name dd2.janlippemeier.de;
 return 301 https://$host$request_uri;
}
```

```
server {
 server_name dd2.janlippemeier.de;
 listen            443 ssl;
 ssl_certificate /etc/letsencrypt/live/janlippemeier.de/fullchain.pem;
 ssl_certificate_key /etc/letsencrypt/live/janlippemeier.de/privkey.pem;
 location / {
  proxy_pass http://localhost:5000/;
  proxy_set_header Host          $host;
  proxy_set_header X-Real-IP     $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Client-Verify SUCCESS;
  proxy_set_header X-Client-DN   $ssl_client_s_dn;
  proxy_set_header X-SSL-Subject $ssl_client_s_dn;
  proxy_set_header X-SSL-Issuer  $ssl_client_i_dn;
  proxy_set_header X-Forwarded-Proto https;
  proxy_read_timeout 1800;
  proxy_connect_timeout 1800;
 }
}
```

Listing 1: Auszug Nginx Konfiguration

Mit dieser Konfiguration wird mit dem ersten Server-Block bewirkt, dass sämtliche Anfragen über HTTP auf das sicherere Protokoll HTTPS umgeleitet werden. In dem zweiten Server Block wird dann die Funktion von Nginx als Reverse-Proxy genutzt. Ein Reverse-Proxy ist ein Server, der Anfragen eines Clients entgegennimmt und nach einer internen Definition an andere Server weiterleitet. Die Antworten dieser, für den Client unbekannten, Server werden von dem Reverse-Proxy wieder an den Client zurückgegeben. Für weitere Details wird auf die offizielle Webseite von Nginx zu diesem Thema verwiesen [1] Durch diesen definierten Reverse-Proxy werden alle Anfragen an diese Subdomain intern an den Port 5000 weitergeleitet. Der Port 5000 muss somit nicht von außen zugänglich sein.
Zusätzlich wurde aus Sicherheitsgründen mit *ssl_certificate* und *ssl_certificate_key* ein Wildcard SSL Zertifikat angegeben, welches eine SSL-Verschlüsselung für alle Subdomains ermöglicht.

Siehe zur Funktionsweise der verschiedenen verwendeten Nginx-Instanzen folgendes Diagramm:

[1] Nginx Reverse Proxy Dokumentation

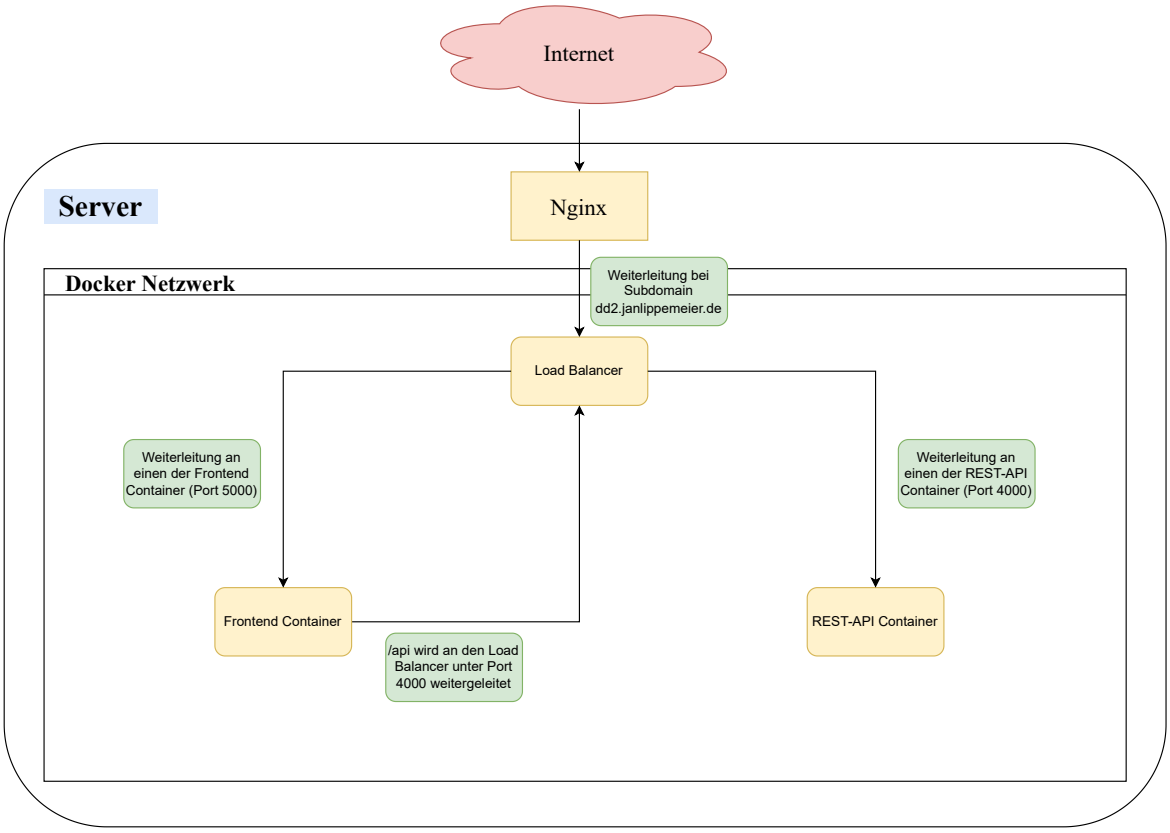

Abbildung 2: Nginx Konfigurationen

Mit diesem Diagramm wird gezeigt, dass Requests an die Sub-Domain dd2.janlippemeier.de an das Docker-Netzwerk [18] weitergeleitet werden. Dort werden sie entsprechend vom Load-Balancer an den Frontend-Container weitergeleitet. Dieser gibt zum einen die HTML Seiten zur Darstellung im Browser zurück. Zum anderen gibt der Frontend-Container Daten von Requests an die API zurück, indem die Reverse-Proxy Funktionalität von Nginx genutzt wird, um die Daten über den Load-Balancer von der REST-API abzufragen.

4.2 Datensicherung

Auf dem Server wurde mittels Cron [17] ein, sich in regelmäßigen Intervallen wiederholender Job, nachfolgend als Cronjob bezeichnet, zur Datensicherung eingerichtet. Dieser Cronjob wird täglich ausgeführt und sorgt für die Erstellung eines Backups der Datenbank (vergleiche Kapitel 6). Da es sich um ein unkritisches studentisches Projekt handelt, ist diese Datensicherung als Machbarkeitsbeweis anzusehen. Somit wird es nicht gemäß eines rationalen Best-Practices auf einem anderen System gespeichert.

4.2.1 Backup Erstellung

Zur Installation von postgres [22] und somit des Befehls *pg_dump* auf dem VPS wird der folgende Befehl verwendet:

```
sudo apt install postgresql
```

Listing 2: Befehl zum Installieren von PostgreSQL

pg_dump ist ein Kommando-Zeilen-basierter Befehl zum Erstellen von Backups. Um das Passwort zu hinterlegen wird eine Umgebungsvariable mit dem Namen *PGPASSWORD* erstellt, die das Passwort für den Admin Nutzer enthält. Die Erstellung eines Backups erfolgt über den Befehl:

```
pg_dump
    -h localhost
    -U postgres
    DB_WATER
    | gzip
    > "/Backup/DD2/backup‘date +\%Y-\%m-\%dT\%H:\%M:\%S‘.gz"
```

Listing 3: Backup Befehl

Durch den folgenden Befehl wird der von *pg_dump* zurückgelieferte String unter Verwendung von *gzip* komprimiert.

```
    | gzip >
```

Listing 4: Komprimieren des Backups

Dieser komprimierte String wird in eine Datei mit dem im Befehl nachfolgend spezifizierten Dateinamen geschrieben.
Der mit */Backup/DD2/backup‘date +%Y-%m-%dT%H:%M:%S‘.gz”* spezifizierte Dateiname besteht aus dem Wort *backup* gefolgt von dem aktuellen Zeitstempel und der Dateiendung *.gz*, die anzeigt, dass die Datei mittels gzip komprimiert wurde. Diese Datei wird in dem Ordner mit dem Pfad */Backup/DD2* gespeichert.

4.2.2 Einrichtung von Cron

Cron wurde auf dem VPS mittels des folgenden Befehls installiert:

```
    sudo apt install cron
```

Listing 5: Befehl zum Installieren von cron

Cron ist ein Tool zum Anlegen der genannten Cronjobs. Damit regelmäßig die Cronjobs ausgeführt

werden können, wird mittels des Befehls *sudo systemctl enable cron* die Ausführung im Hintergrund erlaubt. Durch den folgenden Befehl wird ein neuer Cronjob angelegt:

```
crontab -e
```

Listing 6: Befehl zum Erstellen eines neuen Cronjobs

Um die Häufigkeit der Ausführungen zu definieren wird die folgende Definition verwendet, die dafür sorgt, dass der danach eingegebene Befehl zum Erstellen eines Backups täglich um 04:00 morgens ausgeführt wird:

```
0 4 * * *
```

Listing 7: Definition des Ausführungsintervalls

5 Docker (J.L.)

Die Kerneigenschaft von Docker [6] besteht darin, dass Anwendungen in sogenannte Docker-Container gekapselt werden. Dadurch können Docker-Container unter geringem Aufwand auf Windows, Linux und Macintosh-Systemen ausgeführt werden. Diese Docker-Container stellen die Funktionalität eines eigenen Linux-Systems zur Verfügung. Unter wenigen Ausnahmen, die geringfügige Anpassungen benötigen, ist eine hohe Portierbarkeit zwischen unterschiedlichen Betriebssystem gegeben. Dies ist insbesondere von Gewicht, da im Team unter Windows 10 und 11 auf mehreren PCs gearbeitet wurde und die Produktiv-Version des Projekts auf einem Linux-System gehostet wird. Docker-Container weisen zwar Ähnlichkeiten zu virtuellen Maschinen auf, unterscheiden sich jedoch in folgenden Punkten von diesen:

1. Andere Virtualisierungsstrategie (Direkt auf dem Kernel)
2. Geringerer Speicherverbrauch
3. Optimierung für das Aufsetzen einer Micro-Service-Infrastruktur

5.1 Docker-Compose

Docker-Compose [21] ist ein Tool mit dem unter geringem Aufwand eine Micro-Service-Architektur erstellt werden kann. Zur Konfiguration wird eine *.yml* Datei geschrieben, in der definiert wird, welche Docker-Container mit welchen Parametern gestartet werden sollen. Die Docker-Container befinden sich in ihrem eigenen Netzwerk [18], welches von außen ohne weitere Konfigurationen nicht zu erreichen ist. Das Netzwerk ist über Ports des Host-Systems, die zu Ports von den Docker-Containern zugeordnet werden, zu erreichen. Somit kann unter anderem ein anderer Port nach außen verwendet oder aber Ports von Containern, zum Beispiel von einer Datenbank, von Zugriffen aus anderen Netzwerken geschützt werden. Innerhalb des definierten Netzwerkes können die Container über IP-Adressen oder die in der Konfigurationsdatei definierten Hostnamen ohne Beschränkungen

kommunizieren. Da ein eigenes Netzwerk erstellt wird, ist somit die Portierbarkeit auf andere Systeme deutlich erleichtert.

5.1.1 Docker-Compose Aufbau

Aufgrund der oben beschriebenen Eigenschaften und Umstände wurde die Entscheidung getroffen Docker-Compose für dieses Projekt zu nutzen. Siehe hierzu den folgenden exemplarischen Aufbau der entstanden Docker-Compose-Konfigurationsdatei:

```
version: "3.0"

networks:
  dd2-network:
    driver: bridge

volumes:
  db_mount:

services:
    ...
```

Listing 8: Aufbau der Docker-Compose

Unter dem Punkt *networks* werden die Netzwerke definiert. Es können auch mehrere Netzwerke erstellt werden.
Mittels des gleichnamigen Punktes werden sogenannte *volumes* erstellt. Diese können verwendet werden um eine persistente Speicherung der Daten in einer Datenbank zu gewährleisten. Durch das Stoppen und Entfernen eines Containers werden die darin enthaltenen Daten gelöscht. Docker-Container folgen hier den Prinzipien des Stateless-Programmings [5] und sollten nur in Ausnahmefällen wie beispielsweise von diesem Prinzip abweichen.

5.1.2 Docker-Compose Service Definition

Im Folgenden ist ein Auszug aus dem Punkt services zu sehen, der die Erstellung eines Datenbank-Containers beschreibt.

```
services:
  postgresdb:
    image: postgres
    container_name: dd2-db
    networks:
      - dd2-network
    environment:
      - POSTGRES_PASSWORD=DD2Project
      - POSTGRES_DB=DB_WATER
    ports:
      - '5432:5432'
    volumes:
      - db_mount:/var/lib/postgresql/data
      - ./DB:/docker-entrypoint-initdb.d
    restart: on-failure
```

Listing 9: Datenbank Container

Unter dem Punkt *services* wird hier der Service mit dem Namen postgresdb definiert. Als Grundlage wird das image *postgres* verwendet. Dies ist ein über Dockerhub [7] zur Verfügung gestellter Docker-Image, das eine PostgreSQL Datenbank enthält, die unter geringen Anpassungen lauffähig ist.
Der unter *container_name* spezifizierte Containername ist der Hostname unter dem dieser Container im darunter liegenden spezifizierten Netzwerk erreichbar ist.
Unter dem Punkt *environment* werden Umgebungsvariablen gesetzt, die von dem verwendeten Postgres-Image erwartet werden.
Der Punkt *ports* definiert, welcher dockerinterne Port von außen über welchen Port des Hostsystems zu erreichen ist. Im Kontext dieses Projektes bedeutet dies, dass auf dem Port 5432 des Servers eine Weiterleitung auf den Port 5432 des definierten Services in dem Docker-Netzwerk stattfindet. Diese Portweiterleitung wurde eingerichtet, um einen externen Zugriff für den Entwicklungsprozess auf die Datenbank zu ermöglichen. Diese bewusste Abweichung von den allgemein anerkannten Best-Practices ist damit zu rechtfertigen, dass es sich lediglich um ein studentisches Projekt ohne sensible Daten handelt.
Unter dem Punkt *volumes* wird in dem ersten Eintrag festgelegt, dass die Daten, die Postgres abspeichert, persistent auf dem Host-System abgelegt werden. Im zweiten Eintrag werden sämtliche Inhalte des Ordners *DB*, der in dem Repository des Projektes liegt, in den Ordner */docker-entrypoint-initdb.d* kopiert. In diesem Ordner sollen Skripte abgelegt werden, die, insofern noch keine Datenbanken vorhanden sind, beim Start ausgeführt werden. Im Ordner *DB* liegt eine Datei mit dem Namen *init-user-db.sh*, die Anweisungen zum Erstellen von Datenbankbenutzern enthält.
Der letzte Eintrag *restart* bewirkt, dass bei einem unvorhergesehenem Fehlerfall dieser Docker-Container neu gestartet wird.

5.1.3 Micro-Service-Architektur

Das Projekt wurde als Micro-Service Architektur entwickelt. Daher bedarf es mehrerer Servicedefinitionen in der Docker-Compose-Konfigurationsdatei. Dise Servicedefinitionen erfolgen unter den folgenden Namen:

1. postgresdb
2. pipeline
3. api
4. web
5. load

Unter dem Punkt *pipeline* wird ein Container definiert, der die Daten-Pipeline umfasst. In diesem werden die Daten von der *PegelOnline* REST-API geladen, bearbeitet und in der Datenbank gespeichert. Dieser Prozess wird zyklisch wiederholt.
Der Punkt *api* definiert die Buildanweisungen für einen Docker-Container, der die REST-API enthält, die die Daten aus der Datenbank für das Frontend bereitstellt.

Der Punkt *web* enthält die Anweisungen für das Frontend, welches die REST-API konsumiert und die Daten ansprechend aufbereitet.
Die beiden zuletzt genannten Services können mit mehreren sogennanten Replicas (als Klone oder Kopien zu verstehende Container) erstellt werden, um eine höhere Verfügbarkeit und Performance zu gewährleisten.

5.2 Load-Balancer

Aufgrund der oben beschriebenen Replicas ist ein Load-Balancer notwendig. Dieser wird im Punkt load definiert. Der Load-Balancer sorgt dafür, dass die Anfragen an die zugehörigen Services unter einer Aufteilung der Last weitergeleitet werden. Dafür wird die folgende Konfigurationsdatei verwendet:

```
user  nginx;

events {
    worker_connections 1000;
}
http {
    server {
            listen 4000;
            location / {
                proxy_pass http://api;
            }
    }
    server {
            listen 5000;
            location / {
                proxy_pass http://web;
            }
    }
}
```

Listing 10: Load-Balancer Nginx Konfiguration

Diese Konfiguration ist notwendig, da mehrere Frontend- und REST-API-Container erstellt werden. Die Netzwerkkonfiguration von Docker erlaubt jedoch nur das Port-Binding an einen bestimmten Container. Über Nginx kann hier die Funktionalität erreicht werden, dass das Frontend oder die REST-API angesprochen werden können als ob sie jeweils nur ein einzelner Container wären.

5.3 Verbindung Frontend - REST-API

Zur Anbindung des Frontends an die API wurden sogenannte environments [2] verwendet. Diese sind ein vereinfachendes Konzept von Angular [8] um unterschiedliche Variablenwerte in der Produktiv- und der Entwicklungsumgebung zu verwenden.

5.3.1 Entwicklungsumgebung

Hierbei wurde für die Entwicklungsumgebung mittels folgender Codezeilen auf die Debug-Version der API verwiesen:

```
export const environment = {
  production: false,
  api_url: "http://localhost:8000"
};
```

Listing 11: Entwicklungsvariablen

Dies verweist auf den localhost mit dem Standard-Port, der von FastAPI [10] verwendet wird, um eine Debug-Version der API zur Verfügung zu stellen. Diese führt automatisch einen Neustart durch, sobald zugehöriger Quellcode bearbeitet wurde. Somit können Veränderungen schnell durch das Frontend getestet werden.

5.3.2 Produktivumgebung

Die Verwendung des Docker-Netzwerkes erfolgt in der Produktiv-Version des environments. Dies ist wie folgt definiert:

```
export const environment = {
  production: true,
  api_url: "/api"
};
```

Listing 12: Produktivvariablen

Hier wird für die Variable *api_url* der Wert */api"* angegeben. Dies bedeutet, dass die URL für alle API-Anfragen nach folgendem Schema aufgebaut ist:

https://*Host*/api/*Request_Route*.

Als Host ist in der Produktiv-Version auf dem Server die eingerichtete Subdomain anzusehen also *dd2.janlippemeier.de*. Ein HTTP Request durch den JavaScript Code an */api/measurements/measurements* entspricht also einem HTTP Request an

https://dd2.janlippemeier.de/api/measurements/measurements.

Um nun diese Anfragen, die sich an das Frontend richten, an das Backend weiterzuleiten wurde hier die Reverse-Proxy Funktion von Nginx genutzt. Siehe hierzu den folgenden Ausschnitt der dafür notwendigen Konfigurationsdatei:

```
location /api {
 proxy_read_timeout 3600;
 proxy_pass       http://load:4000;
 rewrite /api/(.*) /$1 break;
 proxy_redirect   off;
```

```
proxy_set_header Host           $host;
proxy_set_header X-Real-IP      $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Client-Verify SUCCESS;
proxy_set_header X-Client-DN    $ssl_client_s_dn;
proxy_set_header X-SSL-Subject  $ssl_client_s_dn;
proxy_set_header X-SSL-Issuer   $ssl_client_i_dn;
proxy_set_header X-Forwarded-Proto https;
proxy_connect_timeout 1800;
}
```

Listing 13: Frontend Nginx Konfiguration

Mit der Definition *location /api* wird bewirkt, dass alle HTTP-Requests an diesen Host, die mit */api* beginnen an die API weitergeleitet werden, sodass über einen HTTP-Request aus dem JavaScript [15] Code an

/api/measurements/measurements

intern an

http://load:4000/measurements/measurements

weitergeleitet wird. Der Hostname *load* wird hierbei genutzt um innerhalb des Docker-Netzwerkes an den Load-Balancer weiterzuleiten

6 Datenbank (D.R.)

In diesem Projekt wird das DBMS von PostgreSQL [22] verwendet. Dieses begründet sich einerseits durch die Kompatibilität mit EF Core [1], ASP.NET [25] und FastAPI [10] andererseits durch die äußeren Faktoren Kosten und Geschwindigkeit.
Das System läuft zusammen mit dem Backend und dem Frontend auf einem Server in einer Dockerumgebung als Container, damit eine dauerhafte Verfügbarkeit gewährleistet werden kann. Genaueres dazu befindet sich in Kapitel 5.

6.1 Modellierung

Der Aufbau der Datenbank richtet sich nach den Prinzipien der Normalisierung. Anhand der semistruktierten JSON [16] (siehe Kapitel 7.1.1) wurden alle Entitäten, Beziehungen und Attribute ermittelt, so dass eine strukturierte Architektur entstanden ist. Diese wurde größtenteils bis zur dritten Normalform gestaltet. Eine Ausnahme bildet die Tabelle *TBL_STATION*. Hier werden aus Gründen der Wartbarkeit und Perfomance zum Beispiel die Attribute *LONGITUDE* und *LATITUDE* in der Tabelle gelassen. Jede Tabelle besitzt aber mindestens einen Primärschlüssel und die Domänen der Attribute sind atomar. Zusätzlich wurden an jede Relation die Attribute *UPDATE_NAME*, *UPDATE_TS*, *INSERT_NAME* und *INSERT_TS* angehangen und die Tabelle *TBL_IMPORT* hinzugefügt. Diese werden für die Fehlerbehandlung in der Pipeline benötigt (siehe Kapitel 7.4.2). Die folgende Übersicht stellt die Beziehungen, Kardinalitäten und Attribute als ER-Diagramm dar.

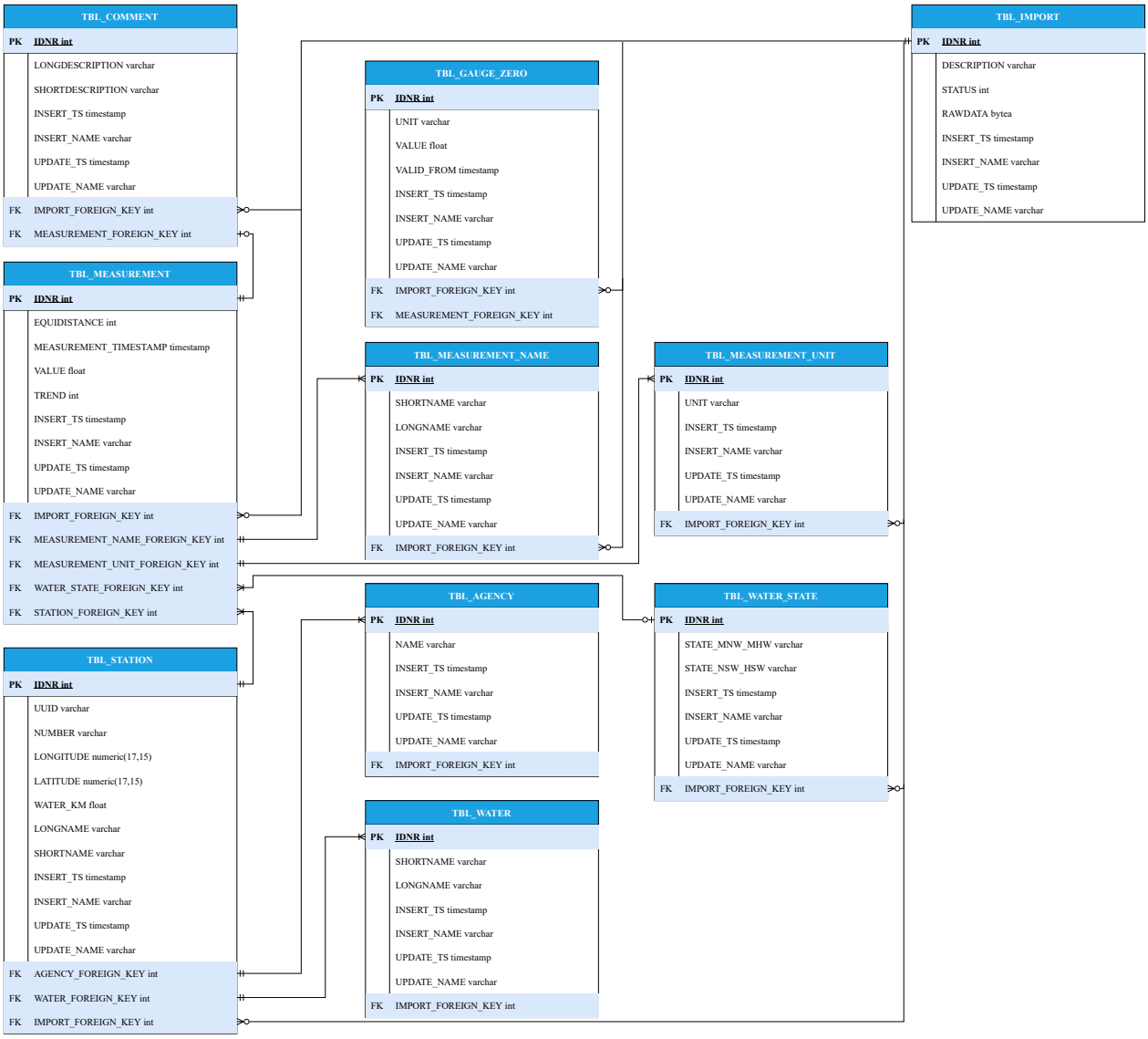

Abbildung 3: ER-Diagramm

Das eigentliche Erzeugen der Tabellen, falls diese nicht vorhanden sind, erfolgt beim Start des Postgres-Containers (vergleiche Kapitel 5). Dort wird automatisch sichergestellt, dass die Datenbank entsprechend dem Schema erstellt und konfiguriert ist. Dazu werden verschiedene SQL-Statements ausgeführt.

```
CREATE TABLE public."TBL_STATION" (
"IDNR" int4 NOT NULL GENERATED BY DEFAULT AS IDENTITY,
"UUID" varchar(255) NOT NULL,
"LONGITUDE" numeric(17, 15) NOT NULL,
"LONGNAME" varchar(255) NOT NULL,
"NUMBER" varchar(255) NOT NULL,
"LATITUDE" numeric(17, 15) NOT NULL,
"SHORTNAME" varchar(255) NOT NULL,
```

```
"WATER_KM" float8 NOT NULL,
"AGENCY_FOREIGN_KEY" int4 NOT NULL,
"WATER_FOREIGN_KEY" int4 NOT NULL,
"IMPORT_FOREIGN_KEY" int4 NOT NULL,
"INSERT_TS" timestamp NOT NULL,
"INSERT_NAME" varchar(255) NOT NULL,
"UPDATE_TS" timestamp NOT NULL,
"UPDATE_NAME" varchar(255) NOT NULL,
CONSTRAINT "PK_TBL_STATION" PRIMARY KEY ("IDNR")
 );
```

Listing 14: Beispiel SQL-Statement zum Erstellen von TBL_STATION

6.2 Trigger und Funktionen

Für jede Tabelle wurde ein Trigger erstellt, der nach einem Update, die in Kapitel 7.4.2 beschriebenen Spalten *UPDATE_NAME* und *UPDATE_TS*, automatisch füllt. Damit soll sichergestellt werden, dass manuelle Änderungen durch Benutzer auf Seiten des Datenbanksystems erfasst werden. Dadurch lassen sich die Ursachen für Anomalien innerhalb der Daten eingrenzen. Außerdem wird für eine höhere Transparenz gesorgt. Um dieses Verhalten mit PostgreSQL zu realisieren muss zusätzlich eine Funktion geschrieben werden, die das Update der Spalten durchführt. Im Gegensatz zu den Triggern reicht eine allgemeine Funktion, weil die Änderungen global in der Variable *NEW* abrufbar sind.

```
CREATE FUNCTION func_update_metadata() RETURNS TRIGGER AS $trg_update_metadata$
    BEGIN
        -- Remember who changed the row
        NEW."UPDATE_TS" := current_timestamp;
        NEW."UPDATE_NAME" := current_user;
        RETURN NEW;
    END;
$trg_update_metadata$ LANGUAGE plpgsql;
```

Listing 15: Funktion für die Aktualisierung der Metadaten

Das Erstellen der Trigger wurde anschließend für alle Tabellen durchgeführt und ist hier für die Tabelle *TBL_STATION* beispielhaft aufgeführt.

```
CREATE TRIGGER trg_update_metadata BEFORE UPDATE ON "TBL_STATION"
    FOR EACH ROW EXECUTE FUNCTION func_update_metadata();
```

Listing 16: Trigger für die Tabelle TBL_STATION

6.3 Konfiguration der Benutzer

Die Gestaltung des Rollenschemas richtet sich nach dem Minimalprinzip. Damit erhält jede Rolle nur die Berechtigungen, die sie benötigt. Insgesamt wurden folgende drei Benutzer angelegt:

- *postgres:* Dieser Benutzer wird beim Erstellen des DBMS von Postgres benötigt und fungiert als Admin. Damit hat er alle Rechte.
- *update_service:* Dieser Benutzer wurde für die Pipeline (siehe Kapitel 7) angelegt und hat die Berechtigungen:
  (a) SELECT
  (b) UPDATE
  (c) DELETE
  (d) INSERT
  (e) CREATE
  (f) DROP
  (g) ALTER
- *api_service:* Dieser Benutzer wurde für die API-Schnittstelle erstellt und hat folgende Berechtigungen:
  (a) SELECT

Wie in der Einleitung dieses Kapitels bereits beschrieben, läuft das Datenbanksystem in einer Dockerumgebung. Damit diese Benutzer reproduzierbar sind und automatisch beim Start eines Containers angelegt werden, wurde ein Skript geschrieben. Dieses wird beim Start des Containers automatisch aufgerufen.

```
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<-EOSQL
    CREATE USER update_service WITH PASSWORD '****';
    GRANT ALL PRIVILEGES ON DATABASE "DB_WATER" TO update_service;
    CREATE USER api_service WITH PASSWORD '****';
    GRANT CONNECT ON DATABASE "DB_WATER" TO api_service;
    GRANT SELECT ON ALL TABLES IN SCHEMA public TO api_service;
EOSQL
```

Listing 17: Bash zum Erstellen der SQL-Nutzer

6.4 Datensicherung

Die Vermeidung von Datenverlust spielt eine wichtige Rolle im Umgang mit Datenbanken. Deswegen wurde in diesem Projekt ein Cronjob angelegt, welcher täglich ein Backup von der Datenbank macht und dieses speichert. Dadurch kann der letzte Stand wiederhergestellt werden. Die Verwaltung der Hardware findet auf Seiten der Betreiber des Servers statt. Die Beschreibung der Implementierung befindet sich im Kapitel 5.1.2.

7 Pipeline (D.R.)

Die Pipeline wurde mit der Programmiersprache C# [3] entwickelt. Den Anfang bildet die Schnittstellenanbindung an die REST-API der Website https://www.pegelonline.wsv.de/webservice/ueberblick, die eine Möglichkeit bietet, verschiedene Wasserdaten abzurufen. Die Datenpunkte werden in einem JSON-Format bereitgestellt und in der C#-Applikation auf eine objektorientierte Struktur gemappt. Anschließend werden die Daten ausgewertet und in einen anderen, objektorientierten Aufbau konvertiert. Diese Konvertierung wird benötigt, damit die Objekte entsprechend den Relationen mit dem Entity-Relation-Mapper in eine Postgres-Datenbank geschrieben werden können. Im letzten Schritt wird die Software über Docker bereitgestellt, weil die Software zusammen mit der Datenbank und dem Frontend auf einem Linux-Sever läuft. Außerdem bietet diese Art der Implementierung weitere Vorteile, dazu gehören Sicherheit, gute Portabilität und schnelle Softwarelieferzyklen.

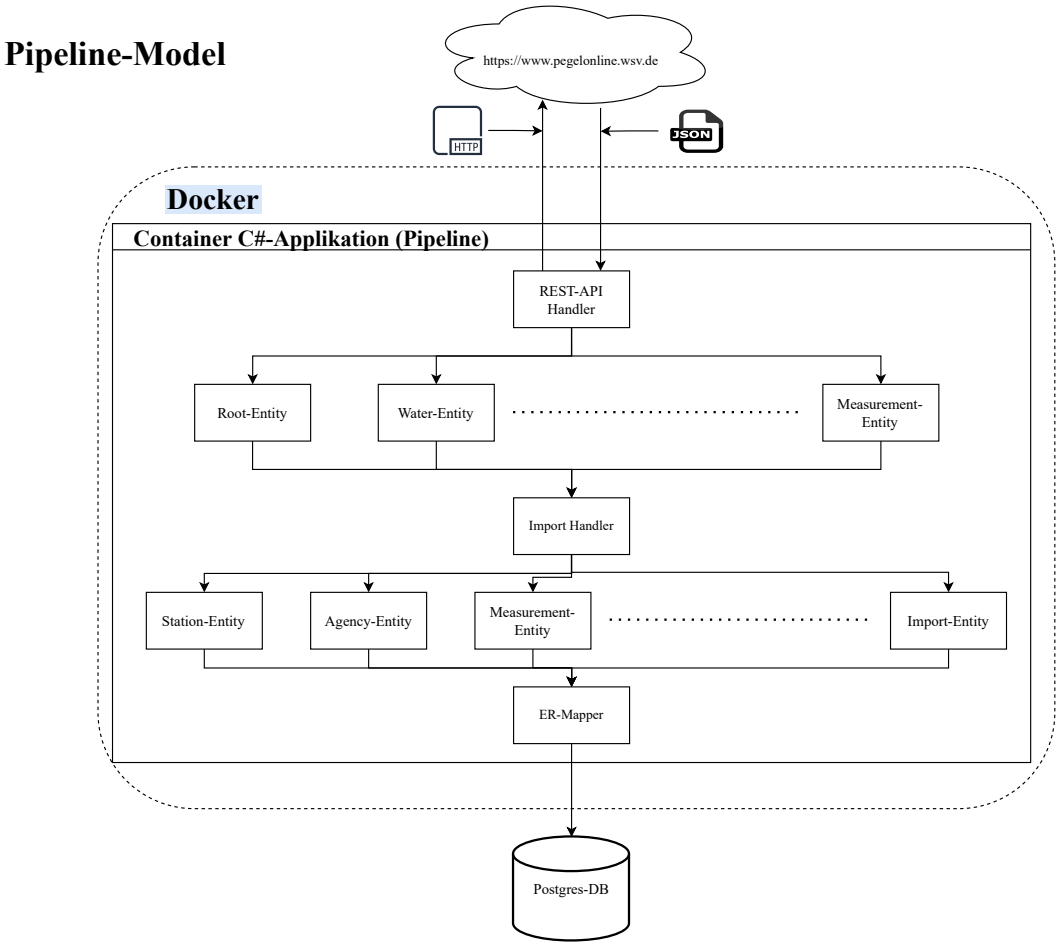

Abbildung 4: Pipeline Modell

7.1 REST-API

Die Wasserdaten werden zyklisch geladen. Das Zeitintervall lässt sich beim Start der Software über eine Umgebungsvariable einstellen und wurde für dieses Projekt auf 10 Minuten gesetzt. Damit die Daten geladen werden können, muss eine Client-Verbindung aufgebaut und ein HTTPS-Request gesendet werden. Dieser Request ist parametrisierbar, um die gewünschten Daten in einem bestimmten Format zu erhalten. Für dieses Projekt wurde auf die stations.json mit folgendem Request zugegriffen:

https://www.pegelonline.wsv.de/webservices/rest-api/v2/stations.json?
includeTimeseries=true&includeCurrentMeasurement=true

Die Parameter *includeTimeseries* und *includeCurrentMeasurement* sorgen dafür, dass die aktuellen Werte der Wasserdaten an die entsprechenden Stationen angehangen werden. Der Aufruf erfolgt in der Applikation über die Klasse *System.Net.Http.HttpClient*, welche den JSON-String entgegennimmt und in die passende Objektstruktur konvertiert. Die konkrete Implementierung befindet sich im Anhang unter Abbildung 28.

7.1.1 Aufbau der JSON

Über den im Abschnitt 7.1 beschriebenen Request wird der JSON-String abgerufen. Für das Projekt sind alle Daten relevant und sehen folgendermaßen aus:

- *uuid - String:* Eindeutige ID der Station.
- *number - String:* Eindeutige Nummer jeder Station.
- *shortname - String:* Abkürzung des Stationsnamens. Oft identisch zu dem ausgeschriebenen Namen.
- *longname - String:* Ausgeschriebener Name der Station.
- *km - Float:* Der Kilometerstand innerhalb des Gewässers.
- *agency - String:* Die Behörde, die zu der Station und den Messwerten gehört.
- *longitude - Float:* Längengrad der Station.
- *latitude - Float:* Breitengrad der Station.
- *water - Dictionary:* Enthält den ganzen Namen und die Abkürzung des Gewässers.
  - *shortname - String:* Abkürzung des Gewässers.
  - *longname - String:* Ausgeschriebener Name des Gewässers.
- *timeseries - List:* Die Liste beinhaltet alle Informationen zu den aktuellen Messwerten.
  - *shortname - String:* Abkürzung des Messwertes.
  - *longname - String:* Ausgeschriebener Name des Messwertes.
  - *unit - String:* Einheit des gemessenen Wertes.
  - *equidistance - Int:* Der zyklische Zeitabstand der gemessenen Werte.

- *currentMeasurement - Dictionary:* Hier werden alle Informationen zu den sich ändernden Messwerten geliefert.
  - *timestamp - String:* Der Zeitpunkt der Messung und die Abweichung zu der UTC-Zeit.
  - *value - Float:* Aktueller Messwert.
  - *trend - Int:* Beschreibt, ob der Messwert fällt (-1), gleich bleibt (0) oder steigt (1).
  - *stateMnwMhw - String:* Setzt den mittleren niedrigsten Wert (Mnw) und den mittleren höchsten Wert (Mhw) in Beziehung. Kommt nur bei Wasserstand-Messwerten vor.
  - *stateNswHsw - String:* Beschreibt das Verhältnis zwischen dem niedrigsten Schifffahrtswasserstand (Nsw) und dem höchsten Schifffahrtswasserstand (Hsw). Kommt nur bei Wasserstand-Messwerten vor.
- *gaugeZero - Dictionary:* Beschreibt die Höhe der Messstation gemessen an einer bestimmten Einheit.
  - *unit - String:* Einheit zu der die Höhe gemessen wurde, zum Beispiel Meter über Null.
  - *value - Float:* Höhe der Messstation.
  - *validFrom - String:* Jahr, Monat und Tag der Messung.

Ein Ausschnitt aus einer Abfrage befindet sich im Anhang unter der Abbildung 19.

7.1.2 JSON zu Entity

Der zurückgelieferte JSON-String wird direkt in eine Liste von Root-Objekten durch den HttpClient (Abbildung 28) konvertiert und die Knotentypen werden durch entsprechende Klassen repräsentiert. Die Attribute sind in den jeweiligen Klassen die Felder. Die Beziehung der Baumstruktur wird durch Felder, welche die erzeugten Objekte der anderen Klassen halten, berücksichtigt. Die nächste Abbildung stellt die Beziehung zwischen dem JSON-Format und den Klassen exemplarisch für Root/Station und Water/Water da.

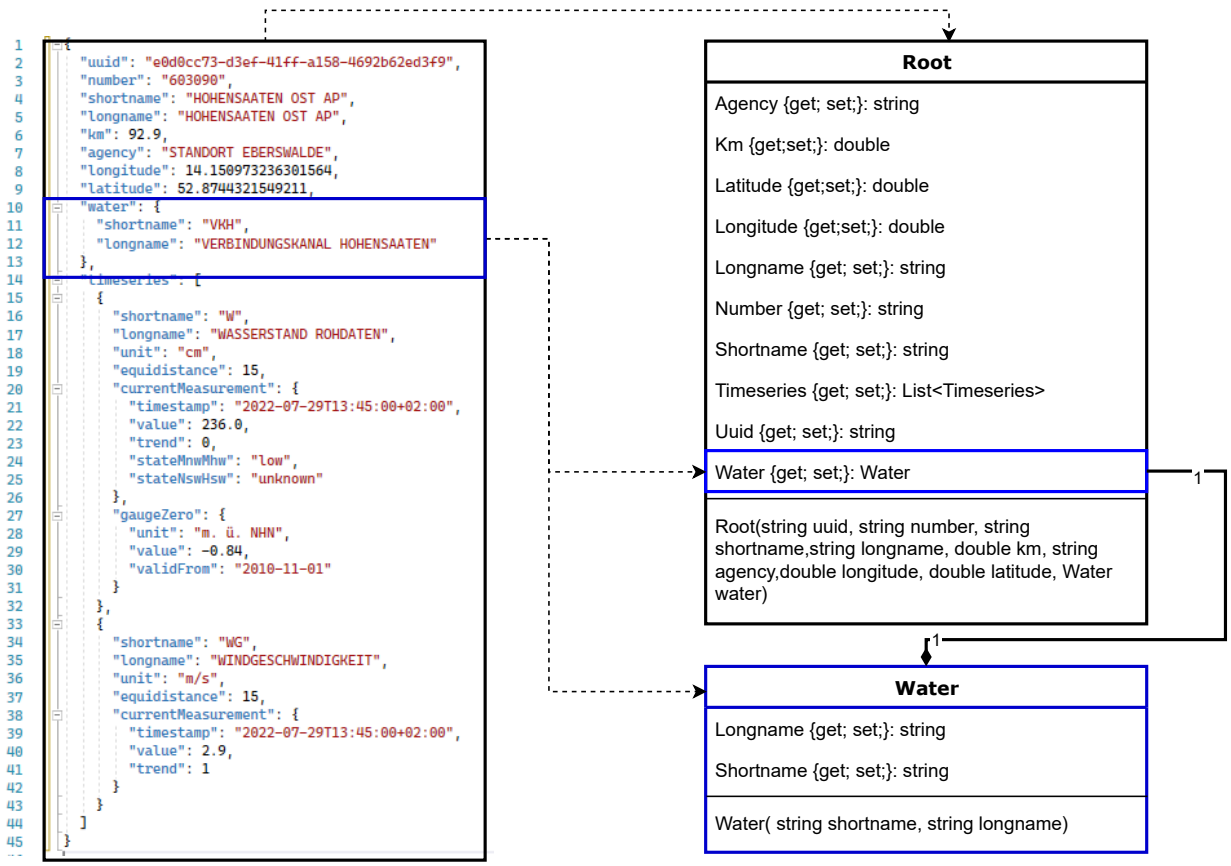

Abbildung 5: Beispiel: JSON zu Entity

7.2 Datenbankanbindung

In diesem Projekt wird die SQL-Datenbank PostgreSQL verwendet. Das Schema (siehe Kapitel 6) für die Daten wurde nach den Prinzipien der Normalisierung gestaltet. Damit die Objekte im Speicher durch die Software in ein konsistentes Datenbankformat konvertiert werden können, wird ein Entity-Relation-Mapper benötigt. Weil die Software in C# entwickelt worden ist, wurde sich für das EF Core [1] von Microsoft entschieden. Das Framework bietet die Möglichkeit, die Datenbank zu modellieren und eine einfache Übersetzung in DML oder DQL basierend auf Listen und Lambda Ausdrücken. Zum Beispiel wird der folgende Zugriff auf eine Liste direkt in ein SQL-Statement übersetzt:

```
var mem = (from c in dbHandler.DbContext.Stations
                    where c.Uuid == root.Uuid
                    select c).FirstOrDefault();
```

Listing 18: Beispiel: Zugriff auf Listen mit EF Core

Im nächsten Listing befindet sich das aus dem Zugriff übersetze SQL-Statement.

```
SELECT * FROM "TBL_STATION" WHERE "UUID" = 'Wert in root.Uuid' LIMIT 1
```

Listing 19: Beispiel: Erzeugtes SQL-Statement

7.2.1 Entity Framework

Das Datenbankschema wird durch die Klasse *DbContext* und Entity-Klassen, welche logisch zu den Relationen gehören, ermöglicht. In DbContext kann beim Start der Software festgelegt werden, welche Relationen erzeugt werden, wie die Kardinalitäten zwischen den Relationen sind, wie die Schlüssel aufgeteilt werden und welche Constraints existieren sollen. In den Entity-Klassen wird der Aufbau der einzelnen Relationen beschrieben.

```
...

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseNpgsql(connectionString);
    }
}
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Station>()
        .HasOne(s => s.Agency)
        .WithMany(c => c.Stations)
        .HasForeignKey(s => s.AgencyForeignKey)
        .OnDelete(DeleteBehavior.Cascade);

    modelBuilder.Entity<Station>()
        .HasOne(s => s.Water)
        .WithMany(c => c.Stations)
        .HasForeignKey(s => s.WaterForeignKey)
        .OnDelete(DeleteBehavior.Cascade);

    modelBuilder.Entity<Measurement>()
       .HasOne(s => s.Station)
       .WithMany(c => c.Measurements)
       .HasForeignKey(s => s.StationForeignKey)
       .OnDelete(DeleteBehavior.Cascade);

    ...
}
```

Listing 20: Beispiel: DbContext erstellen von einem Modell

In der Methode *OnModelCreating* ist exemplarisch das Modellieren von der Relation *TBL_STATION* (siehe Kapitel 6.1) durch die Entität *Station* abgebildet. Durch *.HasOne* und *.HasMany* werden die Kardinalitäten dargestellt. In diesem Modell soll *Agency* n *Station* besitzen und jede *Station* zu

einer *Agency* gehören. Außerdem wird der Fremdschlüssel festgelegt, indem auf das entsprechende Feld in *Agency* verwiesen wird. Als letztes wird der Constraint erzeugt. Damit werden alle *Stationen*, die zu der *Agency* gehören, entfernt, falls die *Agency* gelöscht wird. Die Felder der Klassen geben zusammen mit deren Attributen die Spalten der Relationen in der Datenbank wieder. Außerdem wird durch die Klasse die Relation und deren Namen beschrieben.

```
...

[Table("TBL_MEASUREMENT")]
public class Measurement : EntityBaseImport
{
    private int equidistance = -1;
    [Column("EQUIDISTANCE", Order = 8)]
    [Comment("Distance")]
    public int Equidistance
    {
        get
        {
            return equidistance;
        }
        set
        {
            this.equidistance = value;
        }
    }

    private DateTime measurementTimeStamp = DateTime.Now;
    [Column("MEASUREMENT_TIMESTAMP", Order = 9)]
    [Comment("measurement timestamp")]
    public DateTime MeasurementTimeStamp
    {
        get
        {
            return measurementTimeStamp;
        }
        set
        {
            this.measurementTimeStamp = value;
        }
    }

    ...

    #region Entity relation
    public GaugeZero GaugeZero { get; set; }

    public Comment Comment { get; set; }

    [Required]
    [Column("STATION_FOREIGN_KEY", Order = 9)]
    [Comment("ForeignKey")]
    public int StationForeignKey { get; set; }
    [ForeignKey("Idnr")]
```

```
        public Station Station { get; set; }

        ...
    }
```

Listing 21: Beispiel: Felder und Attribute einer Klasse

Eine komplette Übersicht der Klassenstruktur im Hinblick auf die Vererbung befindet sich im Anhang (siehe Abbildung 20). Jede Klasse hat die oben beschriebenen Felder und Attribute, damit die logische Verknüpfung zu den Relationen hergestellt werden kann. Für weiteren Kontext wird auf die offizielle Seite verwiesen [2].

7.3 Import der Daten

Den Kern des Imports bildet die Klasse *ImportHandler*. Dort werden die beiden Anbindungen (siehe Kapitel 7.1 und Kapitel 7.2) zusammengeführt und der zyklische Abruf der Daten realisiert. Dieses wird durch eine Endlosschleife und eine Zustandsmaschine verwirklicht.

7.3.1 Zustandsmaschine

Die Zustandsmaschine hat die Zustände *LoadRestApiData*, *Update* und *Exception*. Außerdem besitzt sie drei Aktionen. Die erste Aktion ist *Next* und fordert die Zustandsmaschine nach einem erfolgreichen Durchlauf einer der Zustände auf, in den nächsten zu wechseln. Die zweite Aktion ist *Retry*, welche versucht den Zustand zu wiederholen, falls kein kritischer Fehler aufgetreten ist. Die letzte Möglichkeit bietet die Aktion *Exit*. Bei einem kritischen Fehler wechselt die Zustandsmaschine mit der Aktion *Exit* in den Zustand *Exception*, die Endlosschleife wird verlassen und der Import unterbrochen. Im Zustand *LoadRestApiData* werden die Daten wie oben beschrieben (siehe Kapitel 7.1) geladen. Falls dieser Zustand erfolgreich durchlaufen worden ist, wechselt die Zustandsmaschine in den Status *Update*. Hier wird die in Kapitel 7 beschriebene Konvertierung der objektorientierten Strukturen durchgeführt und die Daten überprüft. Dieses wird aus zwei Gründen gemacht: Der erste Grund besteht darin, die Daten auf Plausibilität zu prüfen. Dazu gehören zum Beispiel fehlende Werte oder fehlerhafte Werte, wie ein negativer Breitengrad. Der zweite Grund ist das zyklische Laden der Daten. Laut der Dokumentation werden die Daten alle 15 Minuten aktualisiert. Nach einer Überprüfung stellte sich heraus, dass diese Daten weder zur gleichen Zeit noch genau nach 15 Minuten aktualisiert werden. Deshalb wurde entschieden, die Daten alle 10 Minuten abzufragen und für die verschiedenen Datenpunkte zu überprüfen, ob dieser Eintrag bereits existiert. Dasselbe Konzept gilt natürlich auch für die anderen Strukturen. Als Beispiel können hier erneut die Datenpunkte bezüglich der *Stationen* aufgeführt werden. Aus Gründen der Datenintegrität sollte in der Datenbank nur eine Reihe je Station abgelegt werden. Wenn alle diese Schritte erfolgreich durchlaufen worden sind, dann werden die neuen Datensätze in die Datenbank (siehe Kapitel 7.2) geschrieben und der Zustand wechselt wieder zu *LoadRestApiData*. Damit die 10 Minuten eingehalten werden, wird am Anfang der Schleife überprüft, ob der letzte Start des ersten Zustandes bereits 10 Minuten zurück liegt.

[2] https://docs.microsoft.com/de-de/ef/

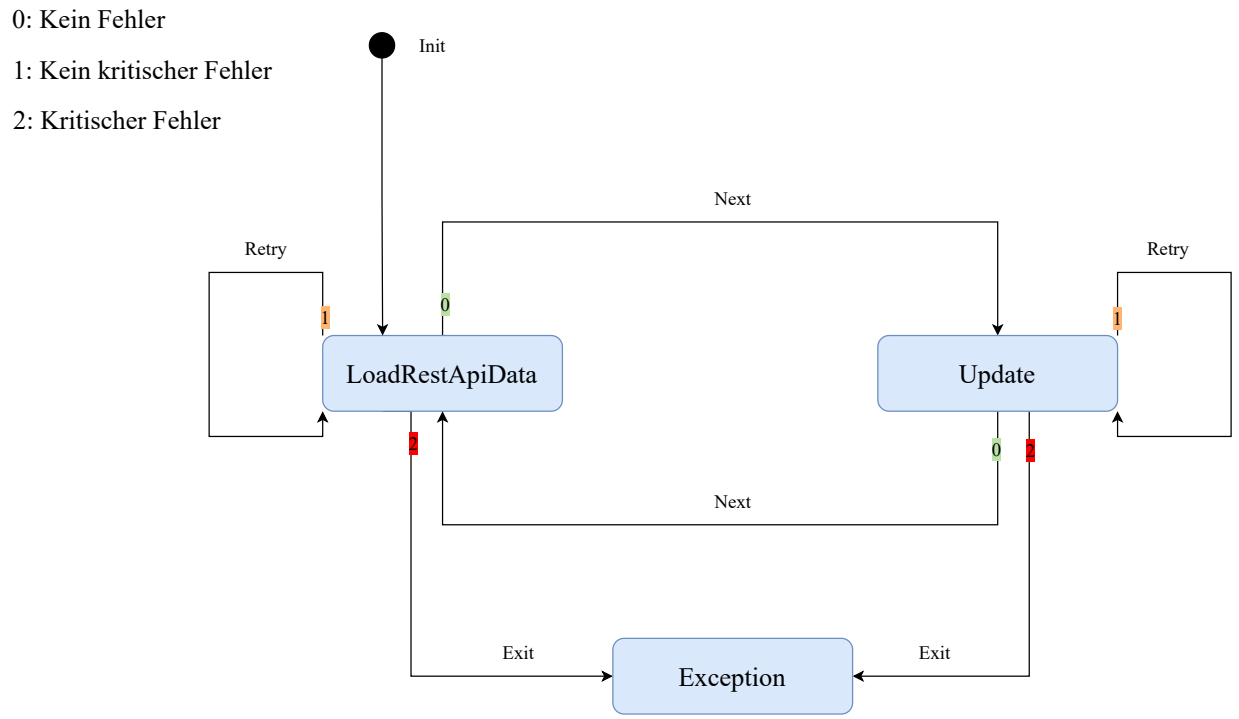

Abbildung 6: Zustandsmaschine des Imports

7.4 Fehlerbehandlung

Die Fehlerbehandlung lässt sich in zwei Kategorien unterteilen. Zum einen das Erkennen oder Abfangen der Fehler und zum anderen das Dokumentieren und Erfassen der Herkunft und Gründe. Das Abfangen der Fehler erfolgt durch gezieltes Ausnahmebehandlungen in der Software, zum Beispiel bei der Verbindung an die Datenbank, und den in Kapitel 7.3.1 angesprochenen Überprüfungen der Sinnhaftigkeit von den gelieferten Daten. Für die zweite Kategorie wird durch Logs und Metadaten die Nachvollziehbarkeit der Fehler sichergestellt.

7.4.1 Logs

In der Software wurde sich für das Paket *Serilog* entschieden, da es eine einfache Möglichkeit bietet, Log-Dateien zu verwalten. Der Hauptnutzen dieser Dateien liegt in der Dokumentation der Fehlerart und Fehlerstelle. Sollte zum Beispiel die Verbindung zu der Datenbank unterbrochen werden und eine Ausnahme wird ausgelöst, dann wird der Fehler, der Zeitpunkt, die Klasse und die Methode in eine Datei geschrieben. Dieses erfordert außerdem einige Anpassungen an das Dockersystem. Damit die Log-Dateien auch persistent sind, wurde ein Volume genutzt, welches auf einen entsprechenden Speicherort gemountet wird.

7.4.2 Metadaten des Imports

Die Metadaten befassen sich mit dem zusätzlichen Speichern von Daten über den Import und dem Verändern der Daten seitens des Datenbanksystems. An jede Relation werden die Attribute *UPDATE_NAME*, *UPDATE_TS*, *INSERT_NAME*, *INSERT_TS* angehangen. In den Spalten *INSERT_NAME*, *INSERT_TS* wird festgehalten, welche Maschine/Nutzer zu welchem Zeitpunkt die Daten eingepflegt hat. In den anderen beiden Spalten werden bei allen Änderungen der Daten in

der Tabelle der Nutzer/Maschine und der Zeitpunkt erfasst. Beim ersten Hinzufügen der Daten sind *INSERT_NAME* und *UPDATE_NAME*, bzw. *INSERT_TS* und *UPDATE_TS* identisch.
Sowohl durch die Logs als auch durch die eben beschriebenen Metadaten wird noch keine Nachvollziehbarkeit der importierten Daten geliefert. Wird durch eine Überprüfung der importierten Daten festgestellt, dass ein Fehler aufgetreten ist, dann lässt sich nur schwer zurückverfolgen, ob die Software falsch programmiert worden ist oder sich die gelieferte Datenstruktur geändert hat. Des Weiteren lassen sich die fehlerhaften Daten im Nachhinein nicht korrigieren. Um diese Punkte zu behandeln gibt es eine weitere Relation *TBL_IMPORT*. Diese wird beim Importieren der Daten selber mit einer *IDNR* und einem Zeitstempel versehen. Außerdem wird der komplette JSON-String als Bytearray in der Spalte *RAWDATA* gespeichert. Zusätzlich erhält jede andere Relation ein Attribut *IMPORT_FOREIGN_KEY*, welches ein Fremdschlüssel auf die *IDNR* in der Tabelle *TBL_IMPORT* ist. Dadurch lässt sich nachvollziehen, wann der Import mit welcher JSON stattgefunden hat und durch den JSON-String selber können Fehler im Nachhinein angepasst werden.

8 REST-API (J.L.)

Um die Daten, welche durch die Pipeline in die Datenbank geschrieben wurden, für das Web-Frontend verfügbar zu machen, wurde eine REST-API implementiert. Die REST-API wurde in Python [12] mittels der Library FastAPI [10] programmiert. Dabei wurde die Library SQL-Alchemy [23] als ORM-Tool zur Anbindung an die Datenbank verwendet.

8.1 FastAPI

FastAPI ist ein Framework zur schnellen Entwicklung von REST-APIs. Es ist ein Open-Source-Projekt. Als Gründe für die Wahl des Frameworks sind anzuführen:

8.1.1 Erfahrungen und Bekanntheit

FastAPI ist ein bekanntes Framework, das in letzter Zeit zunehmend Bekanntheit erlangt hat und nun seine Position gegenüber anderen Frameworks wie z.B. Flask [11] und Django [9] gefestigt hat. Da bereits Erfahrungen mit Flask und FastAPI bestehen lag die Wahl eines dieser Frameworks nahe. FastAPI hat sich nach eigenen Erfahrungen als einfacher in der Handhabung und performanter erwiesen.

8.1.2 Parsen und Validieren

FastAPI bietet das Parsen und Validieren von Daten aus Requests an. Dadurch kann auf manuell implementierte repetitive Überprüfungen verzichtet werden und es kann mit einer grundlegenden Typsicherheit programmiert werden. Dies äußert sich auch in den Rückgaben, die bei der Definition eines Rückgabeschemas diesem zu entsprechen haben. Des Weiteren bietet FastAPI einen einfacheren Zugriff auf Daten aus dem Request, da es über die definierten Keywords den Zugriff auf Path- oder Query-Parameter sowie Inhalte aus dem Request-Body erleichtert.

8.1.3 SwaggerUI und OpenAPI Konformität

FastAPI generiert zu Dokumentationszwecken automatisch eine SwaggerUI [13], die einen Überblick über alle vorhandenen API Routen bietet. Dies ist im folgenden Screenshot der SwaggerUI zu sehen. Zudem können unter geringem Aufwand Definitionen der Rückgabewerte der einzelnen Routen erstellt werden, sodass FastAPI daraus eine OpenAPI-konforme [20] openapi.json Datei generieren

kann. Diese kann zur einfachen Anbindung an einen konsumierenden Service, so beispielsweise auch im Frontend, verwendet werden. Somit kann hier repetitiver und primitiver Programmieraufwand vermieden werden.

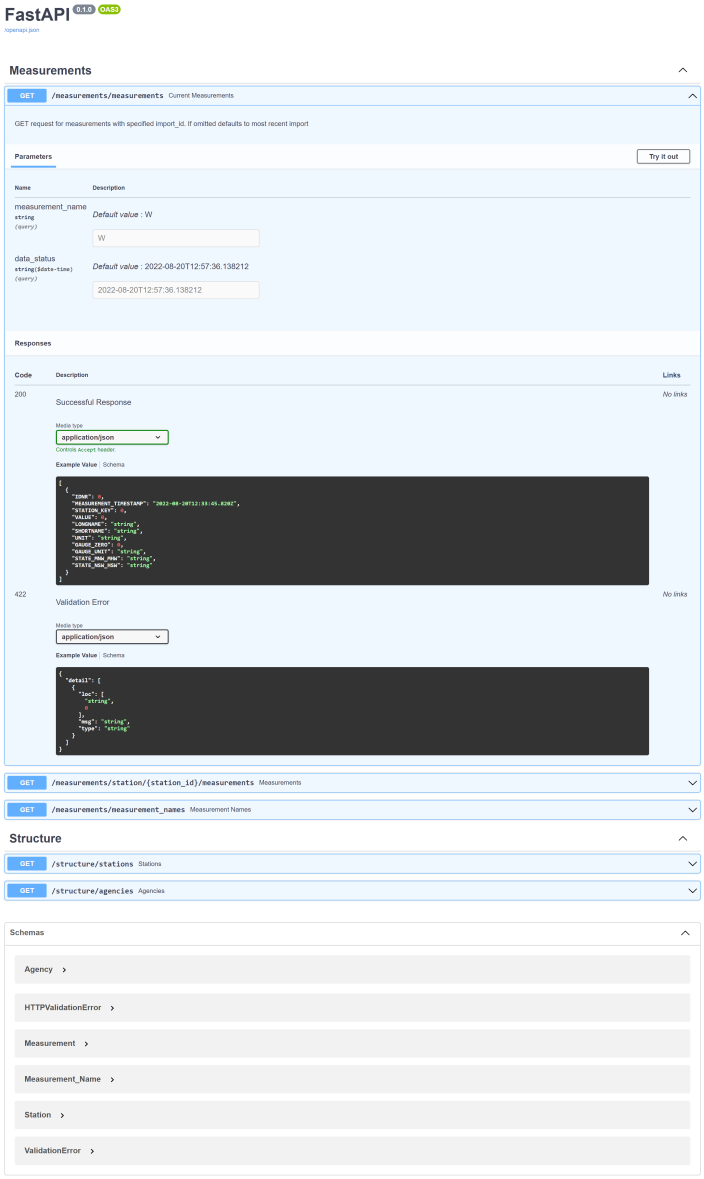

Abbildung 7: Screenshot der SwaggerUI

In diesem Screenshot ist die aus den Angaben automatisch generierte SwaggerUI zu sehen. Besonders hervorzuheben ist hier der ausgeklappte Request, der mögliche Antworten der API schemahaft darstellt.

8.1.4 Strukturierbarkeit der API

Die API kann mit sogenannten Routern in mehrere Teile strukturiert werden. Dadurch ist eine hohe Kohäsion gegeben, da logisch unterschiedliche Aufgaben getrennt behandelt werden können. Zugleich ist auch die Übersichtlichkeit und somit auch die Wartbarkeit erhöht, da die Strukturierung in Form von einer Aufteilung sowohl auf Datei-, als auch Routenebene sowie der Aufteilung in der SwaggerUI vorliegt.

8.2 Aufbau der API

Die API wurde in zwei Router eingeteilt. Routen, die mit dem Prefix /structure beginnen, liefern Informationen über die vorhandenen Stationen und über die vorhandenen Behörden. Routen, die mit dem Prefix /measurements beginnen, liefern Informationen über Messwerte zurück. Dieses Diagramm soll den grundlegenden Aufbau darstellen:

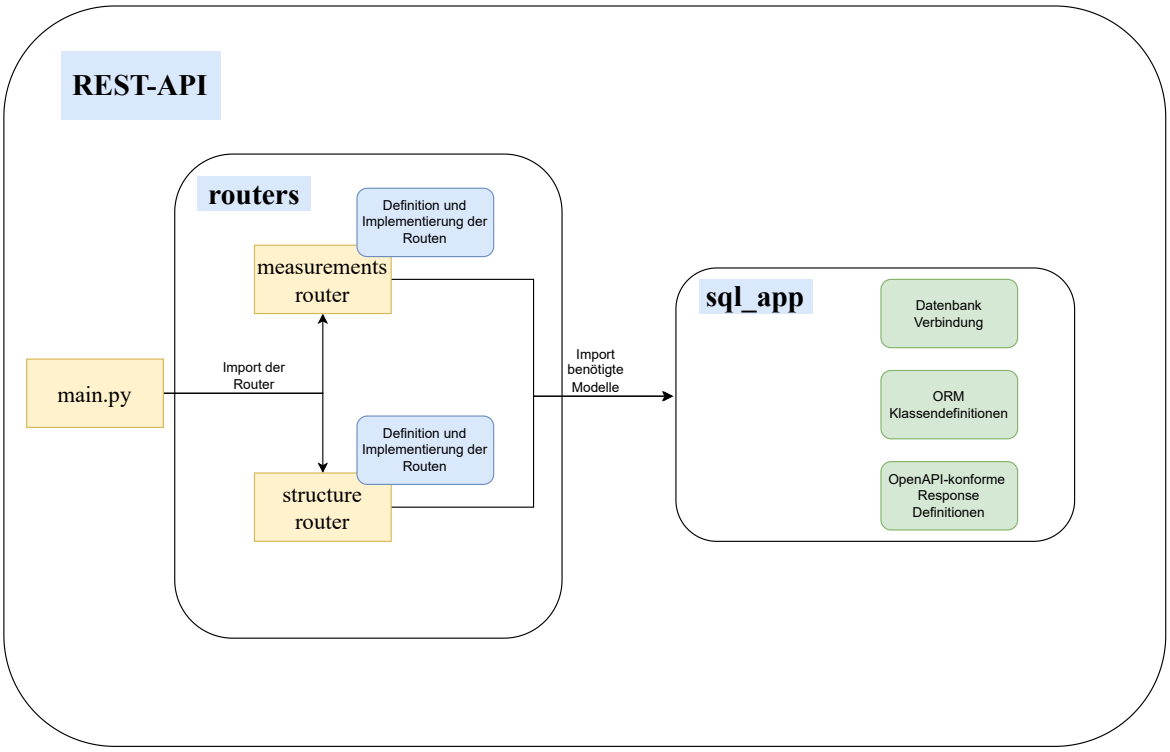

Abbildung 8: Diagram der REST-API

Im folgenden Code Auszug ist die main.py zu sehen, in der die grundlegende Struktur erstellt wird.

```
from fastapi import FastAPI
from starlette.middleware.cors import CORSMiddleware
from routers import measurements, structure
from sql_app import models
from sql_app.database import engine

models.Base.metadata.create_all(bind=engine)
app = FastAPI()
app.include_router(measurements.router,
    prefix="/measurements",tags=["Measurements"])
app.include_router(structure.router,
    prefix="/structure",tags=["Structure"])
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=False,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Listing 22: Aufbau der API

Über die Methode *app.include_router()* wird ein sogenannter Router zur FastAPI Instanz hinzugefügt. Der Prefix bestimmt unter welchem Pfad die Routen aus dem hinzugefügten Router erreichbar sind.
Über die Methode *app.add_middleware()* wird hier erreicht, dass sämtlichen Antworten durch die API Header angefügt werden. Diese sorgen dafür, dass ein Aufrufen der API aus einem Webbrowser heraus (bspw. durch das Frontend) möglich ist und nicht zu Problemen aufgrund der CORS [4] Richtlinien führt.

8.3 Anbindung an die Datenbank

Zum Abfragen von Daten aus der Datenbank wurde SQL-Alchemy verwendet. SQL-Alchemy ist ein ORM-Tool zum Abfragen und Manipulieren von Daten aus einer SQL-Datenbank. Mit dem folgenden Ausschnitt aus dem Quellcode wird eine Klasse erstellt, die eine Tabelle der Datenbank repräsentiert. Dies stellt am Anfang einen Overhead dar, da die zu verwendenden Tabellen, einschließlich ihrer Spalten vorab definiert werden müssen. Jedoch ergibt sich durch das Definieren dieser Klasse eine höherere Wiederverwendbarkeit und eine typsichere Verwendung von Daten aus der Datenbank gegenüber Alternativen, die auf SQL-Statements in Form von Strings und Rückgaben in Form von Listen basieren. Abfragen an die Datenbanken können somit auch sehr einfach und schnell erstellt werden.

```
from sqlalchemy import Column, Integer, String
from .database import Base

class Measurement_Name(Base):
    __tablename__ = "TBL_MEASUREMENT_NAME"

    IDNR = Column(Integer, primary_key=True)
```

```
    LONGNAME = Column(String)
    SHORTNAME = Column(String)
```

Listing 23: Anbindung an die Datenbank

Mittels der oben definierten Klasse kann auf sehr simple Weise für eine Route eine Abfrage erstellt werden. Siehe dazu folgendes Beispiel:

```
@router.get("/measurement_names", response_model=
    list[response_models.Measurement_Name])
def measurement_names(
db: Session = Depends(get_db)):
    return db.query(models.Measurement_Name
    ).with_entities(
        models.Measurement_Name.SHORTNAME,
        models.Measurement_Name.LONGNAME
    ).all()
```

Listing 24: Anbindung an die Datenbank

Die oben gezeigte Abfrage mittels SQL-Alchemy resultiert in einer SQL-Abfrage äquivalent zu der folgenden PostgreSQL-Abfrage

```
SELECT "SHORTNAME", "LONGNAME"
FROM "TBL_MEASUREMENT_NAME"
```

Listing 25: Äquivalente SQL Abfrage

Durch das Verwenden von SQL-Alchemy vereinfacht sich die Erstellung von Rückgaben, da die gefundenen Daten aus der Datenbank über das Key-Value Prinzip adressiert werden können und automatisch sämtliche Datentypen JSON-konform transformiert werden, bevor sie als Response von der API dem Client zurückgesendet werden. Dadurch wird primitiver und oft fehlerbehafteter Programmieraufwand vermieden, indem u.a. Datumswerte in ihre ISO-DateString [14] Repräsentation überführt werden.

8.4 OpenAPI-Konformität der API

OpenAPI [20] ist ein Standard zur Beschreibung einer HTTP-API, der sowohl für Menschen als auch Computer verständlich ist. Der Standard beschreibt die verfügbaren Routen, die erwarteten Input-Daten durch den Client und die zu erwartende Response einschließlich eventueller Datentypen. Der Standard ist unabhängig von der Programmiersprache. Durch die Verwendung des Standards können ansprechende Weboberflächen zur Exploration der Möglichkeiten der API, beispielsweise in Form einer SwaggerUI generiert werden. Ferner kann durch ein geeignetes Tool Quellcode generiert werden, der Methoden, welche Daten aus der API abfragen können, sowie die Typdefinitionen gemäß des Standards umfasst. Dadurch konnte im Frontend ein beträchtlicher Programmieraufwand vermieden werden. Die Typdefinition von JSON-Responses durch die API erfolgte nur einmal in der

REST-API und konnte im Frontend gemäß der äquivalenten Datentypen verwendet werden. Siehe hierzu folgendes Beispiel für die Definition der Klasse Measurement_Name, die als response_model für die oben gezeigte API-Route verwendet wird.

```
from pydantic import BaseModel
class Measurement_Name(BaseModel):
    SHORTNAME: str
    LONGNAME: str
```

Listing 26: Response Model Measurement_Name

9 Web-Framework (H.L.)

Für die Webseite selbst wurde das Web-Framework Angular [8] genutzt. Angular selbst ist ein TypeScript-basiertes Front-End-Webapplikationsframework, welches besonderen Wert auf mobile Platformen legt. Zusätzlich erlaubt es eine objektorierte Programmierung. Das Ziel der Webseite ist es unsere Daten übersichtlich und sinnvoll darzustellen. Dafür wurde ein Dashboard entwickelt, welches mehrere Filteroptionen bietet.

9.1 Module

In Angular erzeugt man einzelne Komponenten, die später als Modul auf der Webseite angezeigt werden können. Unsere erstellten Module unterteilen sich so:

- App-routing Modul
- Dashboard Modul
- Map Modul
- Wettervorhersagen Modul
- Wasserstand/KPI Modul
- Info Modul
- Kontakt Modul
- API Modul

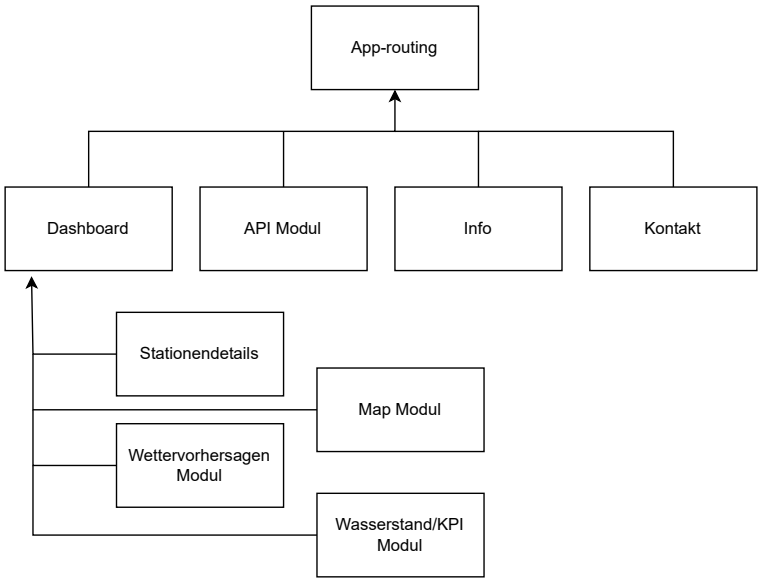

Abbildung 9: Bild des Aufbaus

9.2 App-routing Modul

Das App-routing Modul verwaltet die einzelnen Routen zu den jeweiligen Komponenten, um den Inhalt korrekt dazustellen und zu laden. Dafür haben wir auf die Angular-Router Bibliothek zurückgegriffen, die Methoden zur Verfügung stellt, um ein Routing zu realisieren. Von uns genutzte Klassen sind dabei das RouterModule und Routes. Dies ermöglicht uns eine dynamische Webapp Oberfläche, ohne die Seite neu zu laden bei einer Änderung des Inhalts.

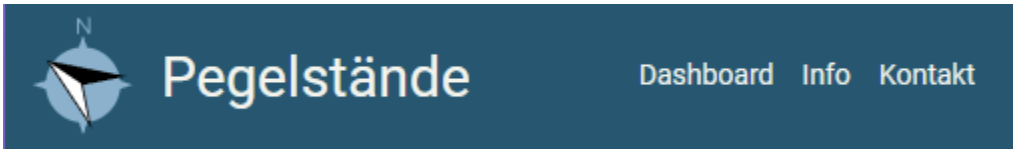

Abbildung 10: Bild der Navigationleiste (Gestaucht)

9.3 Dashboard Module

Unser Dashboard Modul soll die in der Datenbank enthaltenen Daten visualisieren und verständlich erläutern. Dafür nutzen wir globale Filter, über die der Datenstand gewählt werden kann, das gemessene Attribut ausgesucht werden kann und die genauer betrachtete Station ausgewählt werden kann. Zusätzlich zeigen wir Informationen zum letzten Import aus der Datenbank, diese beinhalten den Datenstand und Anzahl der Stationen mit Daten. Bei der Visualisierung von dem Verlauf und den allgemeinen Informationen bietet ein Plotly-Linegraph einen übersichtlichen Einblick in den Verlauf der Daten. Für weitere Informationen geben wir den Namen, die zuständige Behörde, das Gewässer und den Kilometerstand an. Neben Plotly einer JavaScript Open-Source Graphing-Library, haben wir auch Angular Material genutzt, Material Design components for Angular, um graphische Komponenten wie Karten und Auswahlfelder darstellen zu können. Diese Komponenten sind auf Verlässlichkeit und Perfomance getestet.

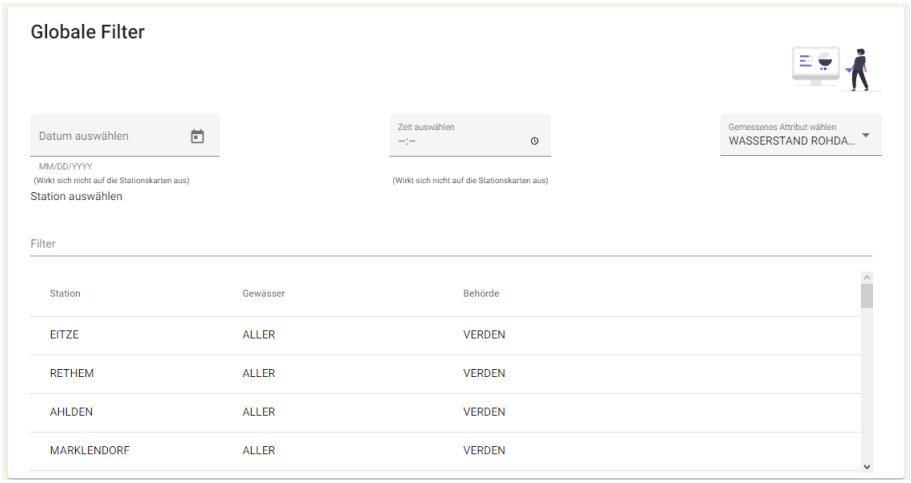

Abbildung 11: Bild der Filteroptionen

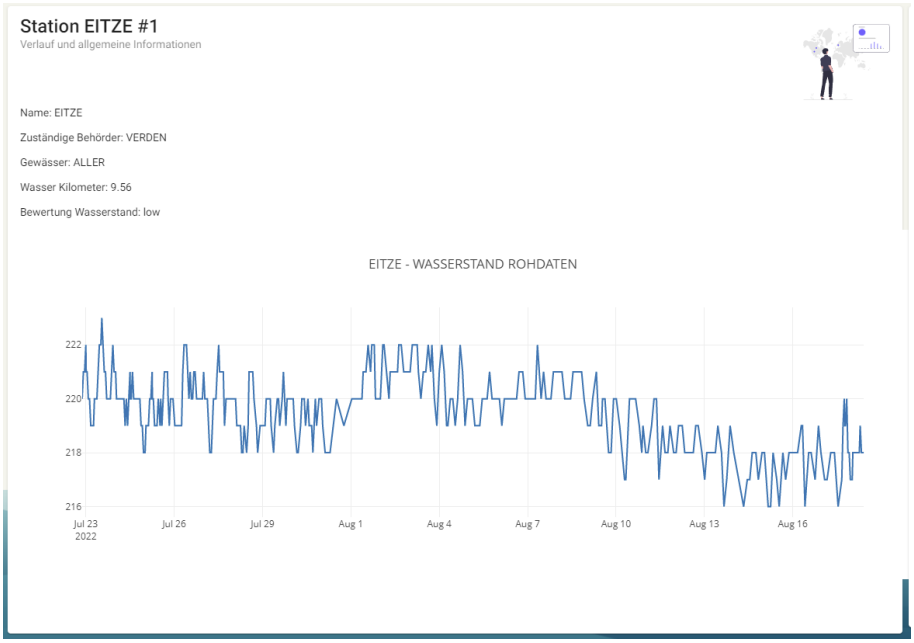

Abbildung 12: Bild des Graphen

9.4 Map Modul

Um auch die Daten der Längen- und Breitengrade nutzen zu können haben wir eine Karte mit Markern integriert, die diese Standorte der einzelnen Stationen anzeigt. Plotly hat uns Kartenfunktionalitäten zur Verfügung gestellt. Für die Kartenteile haben wir OpenStreetMap verwendet, dieses Projekt stellt frei nutzbare Geodaten zur Verfügung (`https://www.openstreetmap.de`). Die einzelnen Marker öffnen ein Popup, welches den Namen der Station offenbart und die Informationen der Station aufruft. Zusätzlich zeigt es auch nur die Stationen an, von denen Daten mit dem passenden Attribut existieren. Die Marker selbst färben sich auch ein je nach dem, wie hoch die Werte sind. So wird ein Überblick über alle Stationen geben, hinsichtlich der aktuellen Lage. Dies hilft zu sehen, ob ein Extremwert nur an einer Station ist oder über mehrere Stationen zu erkennen ist. Dadurch können Trends und Veränderungen der Werte verfolgt werden.

Abbildung 13: Bild der Karte

9.5 Wettervorhersagen Modul

Dieses Modul gibt einem auch noch Auskunft darüber, wie das Wetter werden soll. Dazu wird die API des Deutschen Wetterdienst angesprochen, welche Meteorologische Wetterdaten im Rahmen des Open Data program veröffentlicht (https://www.dwd.de/DE/leistungen/opendata/opendata.html). Wir nutzen diese Chance um auch eine Verbindung zwischen vergangenen Daten aus unserer Datenbank und Prognosen zu schaffen. Dieser Ausblick auf zukünftige Wetterprognosen kann helfen die zukünftigen Daten zu schätzen.

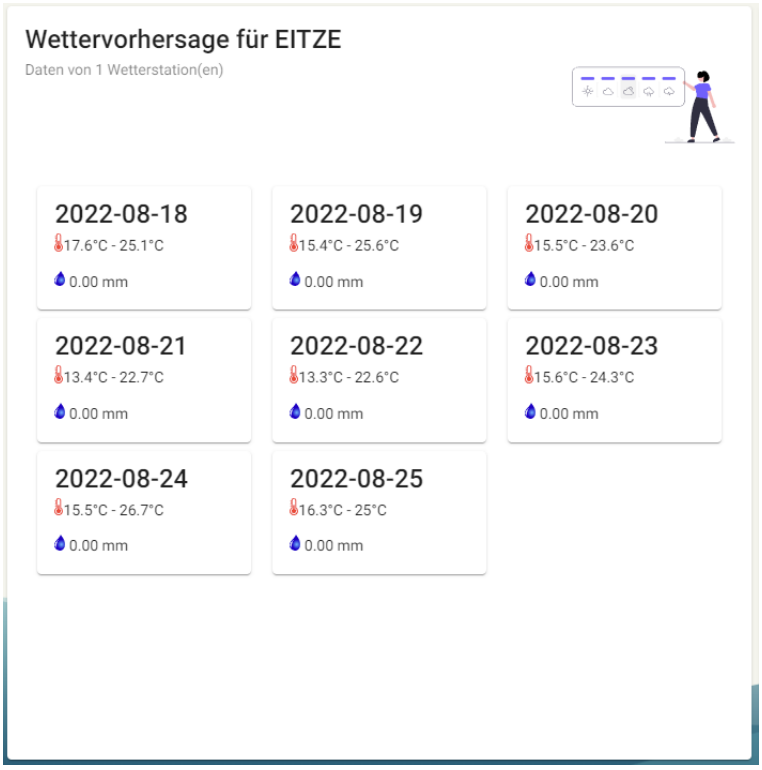

Abbildung 14: Bild der Wettervorhersage

9.6 Wasserstand/KPI Modul

Ein Messdiagramm wird genutzt um den Wasserstand darzustellen, wodurch der aktuelle Wert zwischen dem Minimum und dem Maximum sichbar wird. Zusätzlich wird auch die Veränderung zum vorherigen Wert gezeigt. Farbliche und symbolische Indikatoren werden auch genutzt, um dem User einen schnellen Eindruck zu vermitteln.

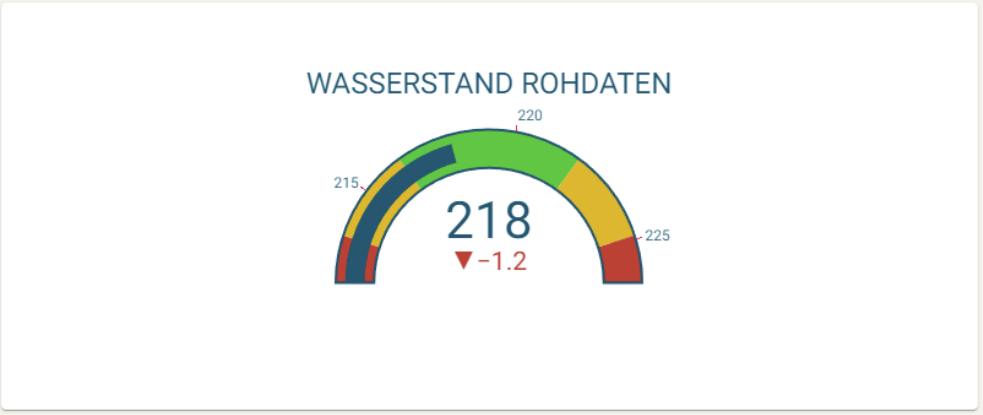

Abbildung 15: Bild der Wasserstandsanzeige

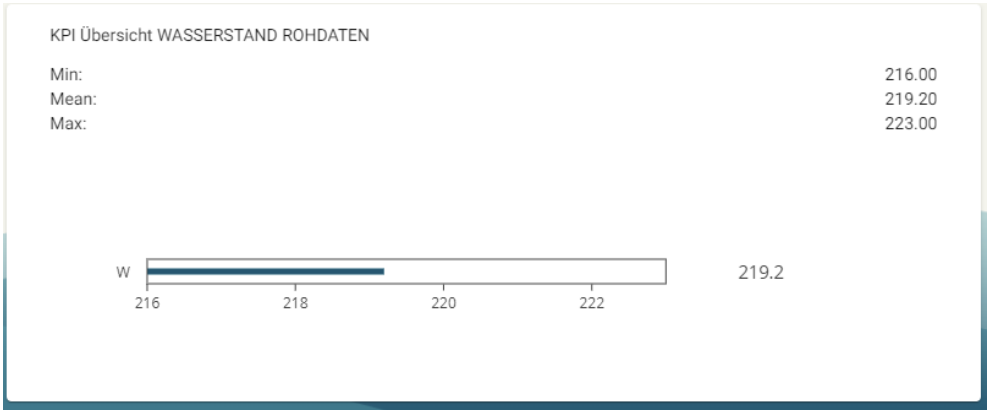

Abbildung 16: Bild der KPI Anzeige

9.7 Info Modul

Hier werden nur Informationen erläutert in welchem Rahmen dieses Projekt entstandt, wer daran beteiligt ist und was die Funktionalität umfassen soll.

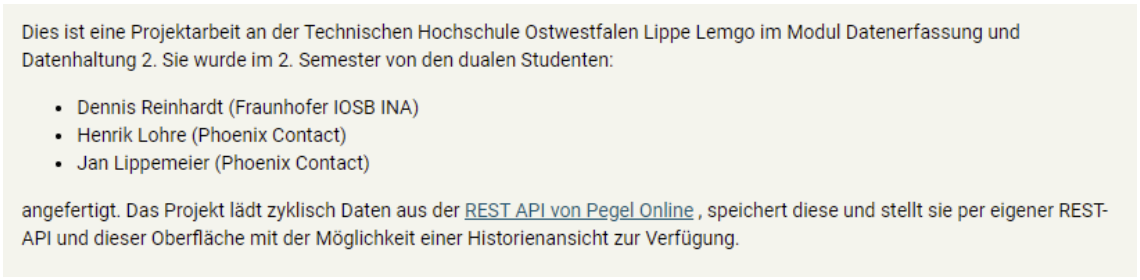

Abbildung 17: Bild der Info

9.8 Kontakt Modul

Hier wurden ein paar Infokarten zu den Entwicklern eingebaut und die zusätzlich die Aufgabeneinteilung abbilden sollen.

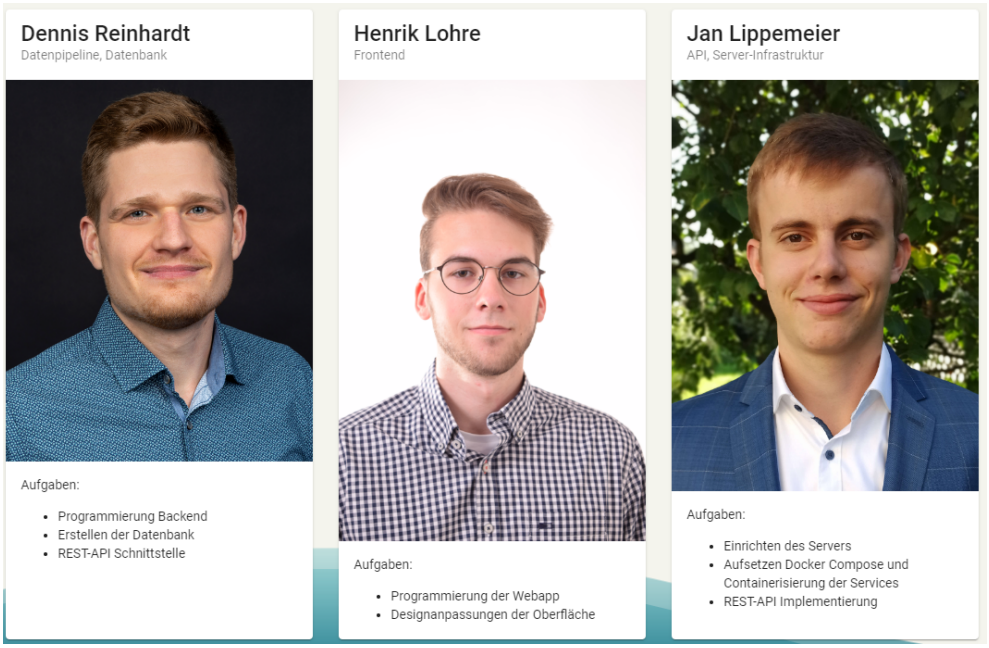

Abbildung 18: Bild der Kontakte

9.9 API Modul

Für die Daten der Datenbank wurde eine REST-API programmiert, die uns verschiedene Routen zur Verfügung stellt. In der Angular Webapp selbst haben einen Service geschaffen der uns Funktionen und passende Interfaces bietet. Jedes Modul kann diese Methoden importieren und nutzen, sodass es für uns einfacher ist zentral Änderungen vorzunehmen. Angular unterscheidet zwischen Komponenten und Services, um die Modularität zu erhöhen und die Wiederverwendbarkeit. Eine Komponente sollte Services für Aufgaben verwenden, die keine Anwendungslogik beinhalten. Services eignen sich für Aufgaben wie das Abrufen von Daten vom Server oder die Validierung von Benutzereingaben.

```
export interface Station {
    AGENCY_NAME: string;
    IDNR: number;
    LATITUDE: number;
    LONGITUDE: number;
    LONGNAME: string;
    SHORTNAME: string;
    WATER: string;
    WATER_KM: number;
  }
```

Listing 27: Beispiel: DbContext eines Interfaces

10 Fazit und Ausblick (H.L.)

10.1 Fazit

Dieses Projekt war für alle Beteiligten ein spannendes Erlebnis. Die Aufgaben, die wir uns selbst stellten, waren spannenden und herausfordernd. Wir haben das Scrum Prinzip angewendet, um uns gegenseitig über unsere Fortschritte zu informieren. Es gab wöchentliche Termine wo wir kurz Probleme oder Ergebnisse präsentierten. Dieses agile Vorgehen half uns schnell auf Änderungen zu reagieren und unterstützte uns bei Problemen, da wir uns gegenseitig halfen. Unser Projekt hat vier große Bausteine geschaffen die es leicht machen das Projekt weiterzuentwickeln. Die Pipeline kümmert sich allein darum unsere Datenbank mit neuen Daten zu versorgen und diese korrekt aufzubereiten. Diese Grundlage schafft qualitative Daten, die wir mithilfe einer REST-API unserer Webapp zu Verfügung stellen. Zusammen haben wir in einem Brainstorming alle Ideen zusammengetragen was unser Dashboard darstellen soll. So entstanden die Graphen, Übersichten und die Karte. Wir wollten so viele Informationen wie möglich übersichtlich dem User zur Verfügung stellen. Hin und wieder gab es mal kleine Kommunikationsfehler oder Bugs im Programm, dies wurde schnell erkannt und entsprechende Fehlerbehebungen wurden zeitnah angewendet. Der jetzige Stand umfasst eine Pipeline, die bnötigte Daten durch eine REST-API von der Webseite abruft und aufbereitet. Die Webapp selbst greift auf eine weitere REST-API zu die Daten aus der Datenbank über verschiedene Routen, in verschiedenen Formen anbietet zu. Diese Daten werden in der Webapp selbst visualisiert, sodass alle relevanten Informationen über die einzelnen Stationen einsehbar sind.

10.2 Ausblick

Für uns ist dieses Projekt nun, im Rahmen des Moduls Datenerfassung und Datenhaltung 2, abgeschlossen. Doch es gibt immer noch Potenzial für Verbesserungen. Schon jetzt erhalten User unserer Webapp einen großen Überblick, darüber welche Daten wo aufgezeichnet werden und wie der zeitliche Zusammenhang ist. Unser Aufbau in die vier Hauptkomponenten, Pipeline, Datenbank, REST-API und Webapp ermöglicht eine einfache Modifizierung oder Erweiterung der Funktionen und Daten. Die Risikobewertung einzelner Wasserstände könnte man noch mit Wettervorhersagen kombinieren, um so zukünftige Entwicklung der Daten zu prognostizieren. Zusätzlich hätte man noch weitere Werte wie Trübung, Chlorid oder Salinität visualisieren und erläutern können.

A C# REST-API Implementierung

```
#region method
/// <summary>
/// The URL field is used to load the current data from the website into a JSON
    format via the RestAPI.
/// The <see cref="HttpClient"/> converts this into the corresponding class
    structure and returns it as <see cref="List{Root}"/>.
/// </summary>
/// <returns><see cref="List{Root}"/></returns>
public List<Root> GetResult()
{
    var t = Task.Run(() => GetResultAsync(Url));
    t.Wait();
    return t.Result;
}

[HttpGet, Route("Results")]
public async Task<List<Root>> GetResultAsync(string url)
{
    // Call asynchronous network methods in a try/catch block to handle exceptions.
    try
    {
        return await httpClient.GetFromJsonAsync<List<Root>>(url);
    }
    catch (HttpRequestException e)
    {
        Programm.LogService.LogError(e);
    }
    return null;
}
#endregion
```

Listing 28: REST-API C# Implementierung

B JSON Beispiel


```
{
  "uuid": "e0d0cc73-d3ef-41ff-a158-4692b62ed3f9",
  "number": "603090",
  "shortname": "HOHENSAATEN OST AP",
  "longname": "HOHENSAATEN OST AP",
  "km": 92.9,
  "agency": "STANDORT EBERSWALDE",
  "longitude": 14.150973236301564,
  "latitude": 52.8744321549211,
  "water": {
    "shortname": "VKH",
    "longname": "VERBINDUNGSKANAL HOHENSAATEN"
  },
  "timeseries": [
    {
      "shortname": "W",
      "longname": "WASSERSTAND ROHDATEN",
      "unit": "cm",
      "equidistance": 15,
      "currentMeasurement": {
        "timestamp": "2022-07-29T13:45:00+02:00",
        "value": 236.0,
        "trend": 0,
        "stateMnwMhw": "low",
        "stateNswHsw": "unknown"
      },
      "gaugeZero": {
        "unit": "m. ü. NHN",
        "value": -0.84,
        "validFrom": "2010-11-01"
      }
    },
    {
      "shortname": "WG",
      "longname": "WINDGESCHWINDIGKEIT",
      "unit": "m/s",
      "equidistance": 15,
      "currentMeasurement": {
        "timestamp": "2022-07-29T13:45:00+02:00",
        "value": 2.9,
        "trend": 1
      }
    }
  ]
}
```


Abbildung 19: JSON Beispiel

C Klassendiagramm

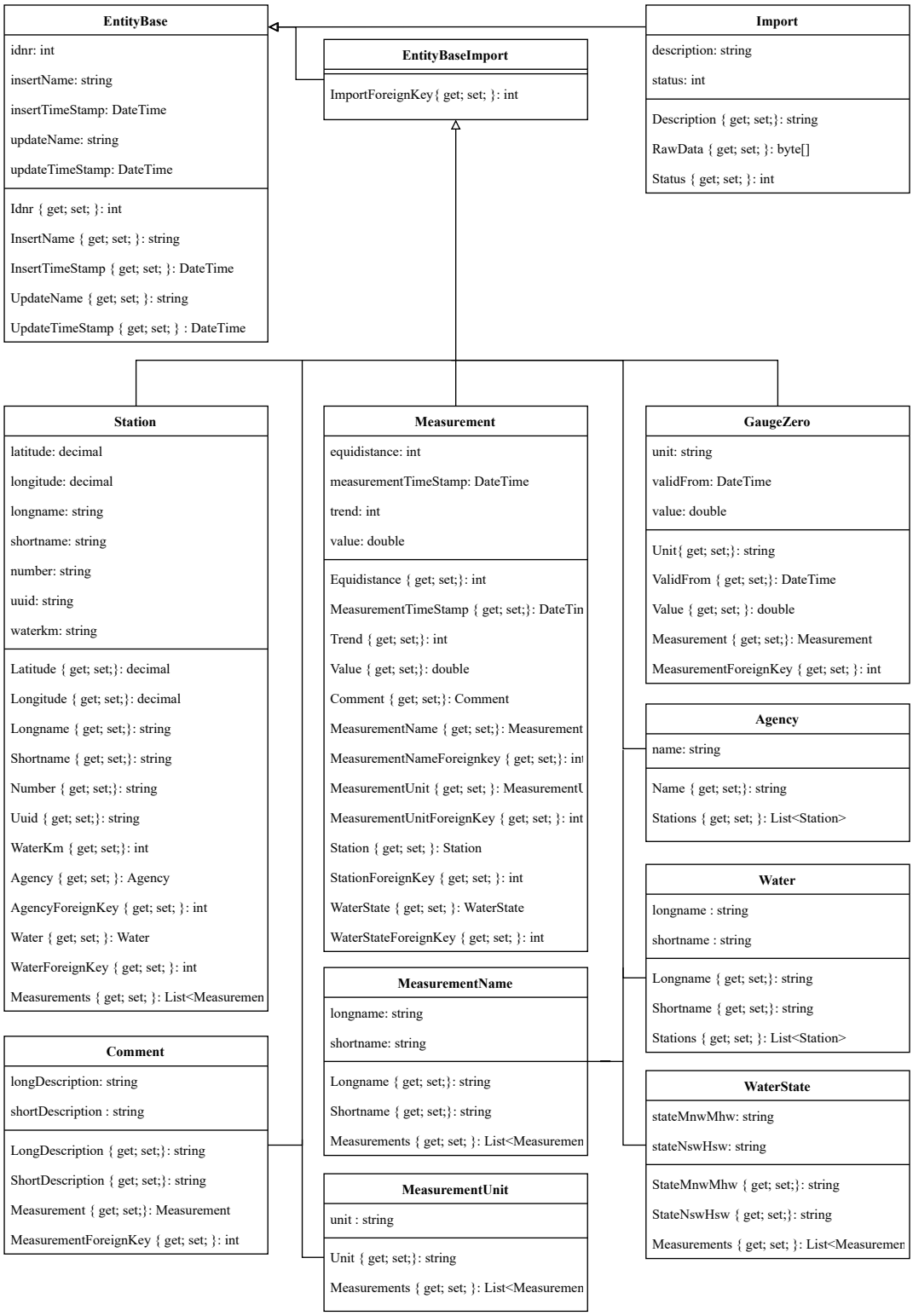

Abbildung 20: Klassendiagramm

D Selbstständigkeitserklärung Jan Lippemeier

Ich erkläre, dass

- alle sinngemäßen Übernahmen aus Arbeiten Dritter mit der Quellenangabe gekennzeichnet sind,
- alle wörtlichen Übernahmen von Textpassagen aus Arbeiten Dritter durch Anführungszeichen und ausführliche Angabe der Belegstelle als Zitat gekennzeichnet sind,
- die vorliegende Arbeit selbständig unter Verwendung der im experimentellen Teil genannten Methoden angefertigt wurde und
- Primärdaten von Experimenten der Arbeit unverändert und in geeigneter Form beigefügt sind.

Ort und Datum Unterschrift

E Selbstständigkeitserklärung Henrik Lohre

Ich erkläre, dass

- alle sinngemäßen Übernahmen aus Arbeiten Dritter mit der Quellenangabe gekennzeichnet sind,
- alle wörtlichen Übernahmen von Textpassagen aus Arbeiten Dritter durch Anführungszeichen und ausführliche Angabe der Belegstelle als Zitat gekennzeichnet sind,
- die vorliegende Arbeit selbständig unter Verwendung der im experimentellen Teil genannten Methoden angefertigt wurde und
- Primärdaten von Experimenten der Arbeit unverändert und in geeigneter Form beigefügt sind.

Ort und Datum Unterschrift

F Selbstständigkeitserklärung Dennis Reinhardt

Ich erkläre, dass

- alle sinngemäßen Übernahmen aus Arbeiten Dritter mit der Quellenangabe gekennzeichnet sind,
- alle wörtlichen Übernahmen von Textpassagen aus Arbeiten Dritter durch Anführungszeichen und ausführliche Angabe der Belegstelle als Zitat gekennzeichnet sind,
- die vorliegende Arbeit selbständig unter Verwendung der im experimentellen Teil genannten Methoden angefertigt wurde und
- Primärdaten von Experimenten der Arbeit unverändert und in geeigneter Form beigefügt sind.

Ort und Datum Unterschrift

Literatur

[1] Ajcvickers. *Überblick über entity Framework Core – EF Core*. URL: https://docs.microsoft.com/de-de/ef/core/.

[2] *Angular Environments*. URL: https://angular.io/guide/build.

[3] BillWagner. *C#-Dokumentation: Einstieg, tutorials, Referenz*. URL: https://docs.microsoft.com/de-de/dotnet/csharp/.

[4] *Cross-origin resource sharing (CORS)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[5] *Definition of stateless*. URL: https://www.pcmag.com/encyclopedia/term/stateless#:~:text=When%20a%20program%20%22does%20not,conditions%20that%20arose%20during%20processing..

[6] *Docker Documentation*. 2022. URL: https://docs.docker.com/.

[7] *Docker Hub*. URL: https://hub.docker.com/.

[8] *Documentation Angular*. URL: https://angular.io/docs.

[9] *Documentation Django*. URL: https://docs.djangoproject.com/en/4.1/.

[10] *Documentation FastAPI*. URL: https://fastapi.tiangolo.com/.

[11] *Documentation flask*. URL: https://flask.palletsprojects.com/en/2.2.x/.

[12] *Documentation Python*. URL: https://www.python.org/doc/.

[13] *Documentation Swagger*. URL: https://swagger.io/docs/.

[14] *ISO 8601 - date and Time Format*. 2020. URL: https://www.iso.org/iso-8601-date-and-time-format.html.

[15] *JavaScript MDN Documentation*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[16] *JSON Documentation W3 Schools*. URL: https://www.w3schools.com/js/js_json_syntax.asp.

[17] *Man Page Crontab*. URL: https://man7.org/linux/man-pages/man5/crontab.5.html.

[18] *Networking overview*. 2022. URL: https://docs.docker.com/network/.

[19] *Nginx Documentation*. URL: https://nginx.org/en/docs/.

[20] *OpenAPI Specification V3.1.0*. URL: https://spec.openapis.org/oas/latest.html.

[21] *Overview of docker compose*. 2022. URL: https://docs.docker.com/compose/.

[22] *PostgreSQL 14.5 documentation*. 2022. URL: https://www.postgresql.org/docs/current/.

[23] *SQLALCHEMY 2.0 documentation*. URL: https://docs.sqlalchemy.org/en/20/.

[24] *Ubuntu Releases*. URL: https://releases.ubuntu.com/20.04/.

[25] Wadepickett. *ASP.NET-Dokumentation*. URL: https://docs.microsoft.com/de-de/aspnet/core/?view=aspnetcore-6.0.