MoboView
Branderkennung mittels KI

DIPLOMARBEIT

Höhere Abteilung für Informatik

01/7/2025 – 26/03/2026

Projektmitglieder: Samuel Riss-Stelzmüllner
Eric Baumgartner
Simon Brandstätter

Betreuer:in: Prof. Michael Stumpfl, Dipl.-Ing.

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Bei der Erstellung der Arbeit haben wir die generativen KI-Tools ChatGPT und Github Copilot zu folgendem Zweck verwendet: Recherche über die Umsetzung von Modelltraining, Debugging.

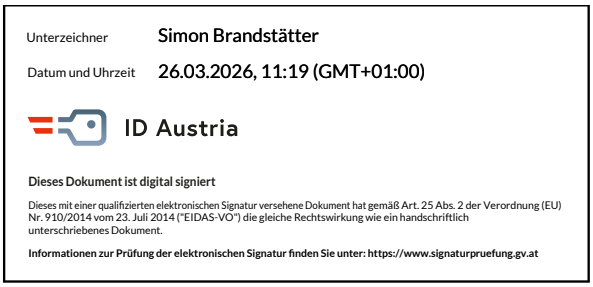
Perg, 26.03.2026

Ort, Datum

Unterschrift, Samuel Riss-Stelzmüllner

Perg, 26.03.2026

Ort, Datum



Unterschrift, Simon Brandstätter

Perg, 26.03.2026

Ort, Datum

Unterschrift, Eric Baumgartner

Danksagung

Wir möchten uns bei der Voestalpine für die absolut reibungslose Zusammenarbeit bedanken. Besonders hervorzuheben ist natürlich die Abteilung für Brandmeldetechnik, in deren Büros wir unser Praktikum absolvieren durften. Die dortigen Arbeitskräfte haben uns in den kurzen vier Wochen unseres Praktikums herzlich aufgenommen und alle Ressourcen zur Verfügung gestellt, um diese Arbeit zu einem vollumfänglichen Erfolg werden zu lassen. Ein besonderer Dank gebührt an dieser Stelle unserem Betreuer von Seiten der Voest, Herrn Buchegger Daniel, welcher uns stets mit Rat und Tat zur Seite stand. Weiters sollten auch die Herren Philip und Philip erwähnt werden, die uns stets bei Laune hielten, wenn diese aufgrund übermäßigen Programmierens im Keller war. Auch Herr Mayerhofer darf bei dieser Gelegenheit nicht unerwähnt bleiben; er beeindruckte uns nicht nur durch seinem brachialen Bizeps sondern auch durch seinen äußerst geschickten Umgang mit dem Autogenschweißgerät. Abschließend möchten wir uns bei Herrn Prof. Stumpfl bedanken; ohne seine Expertise hätten wir es nicht geschafft, die richtigen Schlüsse aus den Projektanforderungen zu ziehen und ein sinnvolles Endprodukt zu entwickeln.

Abstract

The goal of the diploma thesis MoboView is to develop a system for analyzing and monitoring temperature data using modern web and artificial intelligence technologies. Thermal camera data is used to detect potentially critical temperature trends at an early stage and present them in a clear and structured way. A key focus is the automated classification of temperature time series.

The system consists of several main components. The Angular frontend provides the user interface and allows the management and visualization of multiple cameras within a dashboard. Temperature data is displayed in charts and can be analyzed over time.

The backend, implemented with Node.js and Express, handles user, role, and camera management and provides a REST API. It uses a hybrid database architecture: MSSQL for relational data such as users and permissions, and TimescaleDB for efficient storage and querying of time-series data.

A central part of the system is the component, which is based on an LSTM model. This model analyzes temperature sequences and classifies them as `'risky'` or `'not risky'`. This enables the detection of abnormal developments that may indicate potential fire hazards.

Additionally, a proxy server is used to access camera streams and to handle technical constraints such as CORS and authentication. The modular architecture allows all components to work together efficiently and forms a scalable system for intelligent temperature monitoring.

Kurzfassung

Das Ziel der Diplomarbeit MoboView ist es, ein System zur Analyse und Überwachung von Temperaturdaten mittels moderner Web- und -Technologien zu entwickeln. Dabei werden Daten von Wärmebildkameras genutzt, um potenziell kritische Temperaturverläufe frühzeitig zu erkennen und übersichtlich darzustellen. Neben der Visualisierung steht insbesondere die automatisierte Klassifikation von Temperatur-Zeitreihen im Fokus.

Das System besteht aus mehreren zentralen Komponenten. Das Angular-Frontend dient als Benutzeroberfläche und ermöglicht die Verwaltung sowie Anzeige mehrerer Kameras in einem Dashboard. Temperaturdaten werden in Form von Diagrammen visualisiert und können zeitlich analysiert werden.

Das Backend, implementiert mit Node.js und Express, übernimmt die Verwaltung von Benutzern, Rollen und Kameras sowie die Bereitstellung einer REST-API. Es verwendet eine hybride Datenbankarchitektur: MSSQL für relationale Daten wie Benutzer und Zugriffsrechte sowie TimescaleDB für die effiziente Speicherung und Abfrage von Zeitreihendaten.

Ein wesentlicher Bestandteil ist die -Komponente, die auf einem -Modell basiert. Dieses Modell analysiert Temperaturverläufe und klassifiziert diese in `'kritisch'` oder `'unkritisch'`. Dadurch können auffällige Entwicklungen erkannt werden, die auf potenzielle Brandgefahren hinweisen.

Zusätzlich wird ein Proxy-Server eingesetzt, um den Zugriff auf die Kamerastreams zu ermöglichen und technische Einschränkungen wie CORS oder Authentifizierung zu umgehen. Durch die modulare Architektur können alle Komponenten effizient zusammenarbeiten und bilden ein skalierbares System zur intelligenten Überwachung von Temperaturdaten.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ausgangslage 1

1.2 Zielsetzung 1

1.3 Projektumfeld 2

2 Grundlagen und Methoden **4**

2.1 Architektur 4

2.2 Fachbegriffe 27

2.3 Verwendete Entwicklungssysteme 32

2.4 Verwendete Technologien 36

2.5 verwendete bibliotheken und plug-Ins 48

2.6 verworfene Optionen 55

3 Ergebnis **56**

3.1 Architektur 56

3.2 Frontend 57

3.3 Backend 58

3.4 Branderkennungskomponente 58

4 Implementierung **59**

4.1 Branderkennungskomponente 59

4.2 Frontend 71

4.3 Backend 88

5 Aufgabenverteilung **111**

5.1 Samuel Riss-Stelzmüllner 111

5.2 Simon Brandstaetter 112

5.3 Eric Baumgartner 113

6 Resümee **116**

Glossar **V**

Literaturverzeichnis **VII**

Abbildungsverzeichnis **XII**

Quellcodeverzeichnis **XIV**

1 Einleitung

In diesem Kapitel wird das Ziel des Projekts, die dazugehörige Ausgangslage und das Projektumfeld behandelt.

1.1 Ausgangslage

Das voestalpine-Werk mit Standort Linz bedient sich bereits vieler Technologien um Brände ehestmöglich zu identifizieren. Am gesamten Werksgelände sind laut internen Auskünften Stand 2026 etwa 45.000 Brandmelder verschiedener Art verbaut. Weiters hat die Voest durch ihre, am Werksgelände stationierte, Betriebsfeuerwehr die Zeitspanne von der Erkennung des Brandes bis zur Eliminierung minimiert. Diese Maßnahme ist auch begründet, da die 99 der 2023 dort stationierten Mitglieder [1] in selbigem Jahr zu 1141 Einsätzen [2] ausrücken mussten.

1.2 Zielsetzung

Durch die derzeit wachsende Aufmerksamkeit zum Thema KI trat die Frage auf, ob nicht auch die Branderkennung durch ein KI-Modell verbessert werden könnte. Das Projekt soll die Möglichkeiten in diese Richtung in Kombination mit einer Mobotix Kamera (siehe Abschnitt 2.4.1) untersuchen. In Absprache mit dem Auftraggeber wurde dieses Ziel während des Praktikums noch etwas konkretisiert. Die Anwendung soll eine Möglichkeit bieten, mehrere Kameras in einem Dashboard zu verwalten (siehe Abschnitt 4.2). Die, von der Kamera gelieferten, Temperaturdaten sollen mit einem ML-Modell auf mögliche riskante Verläufe kontrolliert werden (siehe Abschnitt 4.1). Die Verwaltung der Kameras und Benutzer soll in einer Backend-Applikation implementiert werden (siehe Abschnitt 4.3).

1.3 Projektumfeld

1.3.1 Projektteam

Das Projektteam setzt sich aus Eric Baumgartner, Simon Brandstätter und Samuel Riss-Stelzmüllner zusammen (siehe Abbildung 1). Alle drei sind Schüler der Höheren Technischen Bundeslehranstalt Perg. Im Rahmen des Projekts setzte Simon seine Expertise in Angular ein, um eine Benutzeroberfläche zu erstellen, Eric nahm sich des Backends an und konnte dort sein Wissen über Datenbanken unter Beweis stellen und Samuel entwickelte ein KI-gestütztes Branderkennungssystem.

Abbildung 1: Eric Baumgartner, Simon Brandstätter, Samuel Riss-Stelzmüllner (von links nach rechts)

1.3.2 Betreuung

Die Diplomarbeit wurde von Herrn Prof. Stumpfl mit tatkräftiger Unterstützung und unglaublichem Fachwissen betreut.

1.3.3 Auftraggeber

Auftraggeber dieser Arbeit ist die Voestalpine (Logo siehe Abbildung 2) mit Hauptsitz in Linz, Oberösterreich. Die Voestalpine ist ein international vertretener Stahlkonzern mit zahlreichen Produktions- und Betriebsstandorten weltweit. Im Rahmen dieser Diplomarbeit fungiert die Voestalpine als Praxispartner und definiert die Anforderungen an die Umsetzung. Damit wird sichergestellt, dass die erarbeitete Lösung einen konkreten Nutzen aufweist.

Im Zuge des vierwöchigen Diplomarbeitspraktikums erfolgte eine tatkräftige Unterstützung durch Daniel Buchegger und Christian Scheer hinsichtlich der Mobotix-Kamera.

Abbildung 2: Voestalpine Logo
[3]

2 Grundlagen und Methoden

In diesem Kapitel werden verwendete Technologien und grundlegende Fachbegriffe erläutert.

2.1 Architektur

2.1.1 Frontend

Architekturüberblick

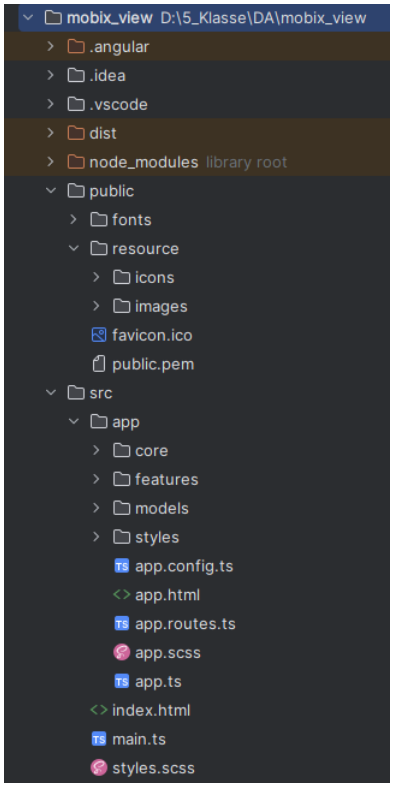


Abbildung 3: Frontend Projektstruktur

Die MoboView Webapplikation folgt einer komponentenorientierten Architektur (Siehe Abbildung 3), die durch das Angular Frontend-Framework (Siehe Abschnitt 2.4.7) vorgegeben wird. Damit kann eine gewisse Wiederverwendbarkeit gewährleistet werden. Es gibt sinngemäß zwei Hauptverzeichnisse, einerseits `'/public'`, welches alle Ressourcen wie Icons, Images, Favicon, uvm. beinhaltet und auf die vom Programmcode aus zugegriffen werden. Das Verzeichnis `'/src'`

enthält hingegen den eigentlichen Quellcode. Der Quellcode unterliegt ebenso einer Hierarchie, wodurch eine saubere Struktur entsteht. Das Verzeichnis `'/core'` stellt die Infrastruktur der Anwendung dar und beinhaltet wiederum die Verzeichnisse `'/api'`, welches die Backend-Endpunkte enthält, sowie `'/guard'` für die Zugriffskontrolle auf die Routen, `'/interceptors'` für die sinngemäße Verwendung des Token-Prinzips und `'/services'` gewährleistet, wenn benötigt eine zentrale Logik. Das Verzeichnis `'/features'` enthält dann die eigentlichen Komponenten bzw. Views. Die Datenstrukturen sind über Interfaces festgelegt, die sich wiederum im `'/models'`-Verzeichnis befinden. Für eine konsistente Formatierung der Webapplikation wurde in `'/styles'` einmalig ein konkretes Styling mittels SCSS (Siehe Abschnitt 2.4.7) umgesetzt.

Assets

Für die Schriftdarstellung kommt eine manuell konfigurierte Chivo-Schriftart zum Einsatz, welche als TTF-Datei, sprich ein vektorbasiertes Format für Schriftarten, gespeichert wird. Diese wurde lokal eingebunden und ist somit unabhängig von externen Diensten.

Die auf der Website verwendeten Icons belaufen sich ausschließlich auf SVG-Grafiken des Anbieters `Lucide Icons`. Bei den sonstigen Bildmaterialien handelt es sich entweder um Dateien, die einer Creative-Common-Lizenz unterliegen oder um eigens erstellte SVG-Grafiken.

Das verwendete Favicon ist eine `18x18` Pixel große Grafik, welche somit akkurat im Browser dargestellt werden kann.

Chart Einbindung

Für die grafische Anzeige von Daten wurde eine Chart-Bibliothek benötigt. Diese wurde mittels

Listing 1: Installation von Chart.js

```
1 npm install chart.js
```

installiert und im Projekt an der dementsprechenden Stelle importiert und eingebunden. (Siehe Listing 2)

Listing 2: Einbindung eines Canvas-Elements

```
1 <canvas id='temperatureChart'></canvas>
```

Die Bibliothek ermöglicht somit eine akkurate Darstellung von Diagrammen direkt im Browser.

Für die Visualisierung der Temperaturwerte wird ein Liniendiagramm verwendet. Die benötigten Messwerte werden aus einem Service geladen (Siehe Abschnitt 4.2.3) und anschließend im Diagramm dargestellt.

Durch die Verwendung von Chart.js (Siehe Abschnitt 2.5.3) können Temperaturverläufe direkt im Browser visualisiert werden.

Models

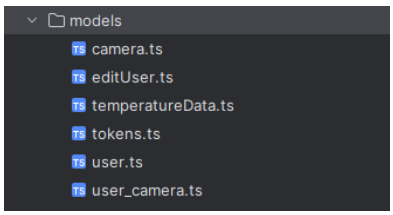


Abbildung 4: Frontend Models

Zur Definition der Datenstrukturen beziehungsweise zur Abbildung der Backend-Daten in TypeScript-Objekten sind Models zum Einsatz gekommen (Siehe Abbildung 4). Zweck der Models ist die Typisierung von Daten in TypeScript. Außerdem wird dadurch auch eine bessere Les- und Wartbarkeit gewährleistet.

Das Model `'camera'` beschreibt eine Kamera und deren Attribute.

Das Model `'temperatureData'` definiert das Format der Messwerte.

Das Model `'user'` repräsentiert einen Benutzer des MoboView Systems und hält die entsprechenden Informationen.

Das Model `'editUser'` kommt für saubere Änderungen am Benutzerkonto zum Einsatz.

Das Model `'tokens'` enthält die Struktur der verwendeten JWTs, die zur Authentifizierung von Benutzern dienen.

Das Model `'user-camera'` beschreibt die Beziehung zwischen Benutzern und Kameras und ermöglicht somit die Zuordnung bestimmter Kameras zu einem Benutzer.

SCSS-Designsystem & Figma Umsetzung

SCSS-Designsystem

Für die Webapplikation existiert ein globales SCSS Styling, welches Schriftart uvm. beinhaltet (Siehe Listing 3).

Listing 3: Globales Styling

```
@font-face {
  font-family: 'Chivo';
  src: url('/fonts/Chivo-VariableFont_wght.ttf') format('truetype');
}

html, body {
  font-family: 'Chivo', 'sans-serif';
  padding: 0;
  height: 100dvh;
  width: 100dvw;
  user-select: none;
}

html, body { height: 100% }
body { margin: 0 }
```

Ebenso wird sich der Vorteil von SCSS zunutze gemacht und globale, sich wiederholende Styling-Attribute in Variablen gespeichert (Siehe Listing 4). Dabei handelt es sich in erster Linie um Farb-Schemata sowie um Kantenabrundungseinstellungen.

Listing 4: Globale SCSS Variablen

```
$primary-color: #000000;
$secondary-color: #FFFFFF;
$tertiary-color: #D9D9D9;

$camera-list-padding: 2.5rem 0rem 2.5rem 0rem;

$border-radius: 0.35rem;
```

Der Zugriff auf die Variablen in den einzelnen Komponenten wird dann durch den Import des globalen `'variables'`-Files in den jeweiligen SCSS-Files bewerkstelligt (Siehe Listing 5).

Listing 5: Variablen Import

```
@use '../../styles/variables' as *;
```

Alle anderen Styling-Vornehmungen sind komponentenspezifisch umgesetzt (Siehe Listing 6) und folgen dabei immer einer hierarchischen Struktur.

Listing 6: Komponenten Styling

```
.container {
  display: flex;
  flex-direction: column;
  width: 24rem;
  height: 100dvh;
  border-right: 3.5px solid $primary-color;
  align-items: center;
  max-height: 100dvh;
  .logo {
    padding: 2rem 1rem;
    height: 2.1rem;
  }
  .input-container {
    display: flex;
    justify-content: center;
    align-items: center;
    border-radius: $border-radius;
    width: 20rem;
    height: 2.5rem;
    border: 3.5px solid $primary-color;

    .magnifying-glass {
      flex: 1;
      height: 1.5rem;
    }
```

Wie wir sehen können, wird vereinzelt immer wieder auf globale Variablen mit dem Schlüssel '$' zugegriffen.

Figma Umsetzung

Die vollständige Webapplikation bzw. das Design des MoboView-Systems wurde zu Orientierungszwecken zuerst vollständig in Figma umgesetzt (Siehe Abbildung 5).

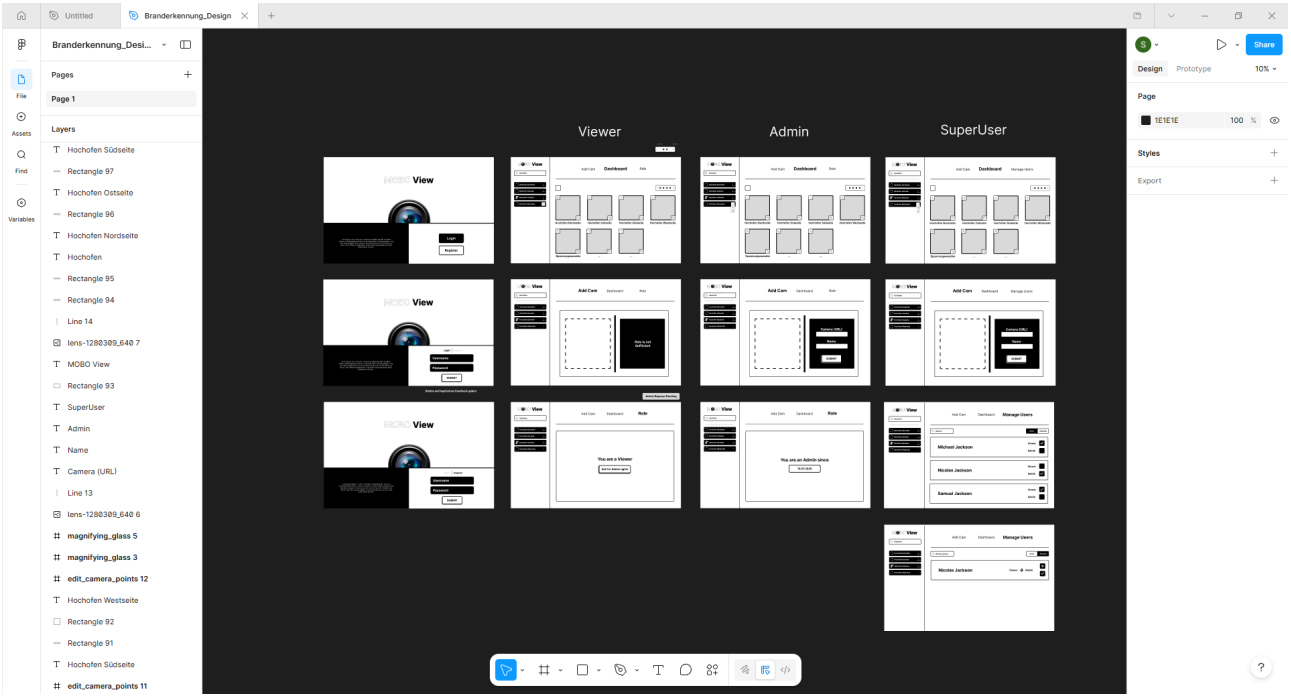


Abbildung 5: Figma Design

Ein Kernelement ist hierbei die Landing-Page (Siehe Abbildung 6) auf welcher man sich in das System einwählen oder einen neuen Benutzer erstellen kann.

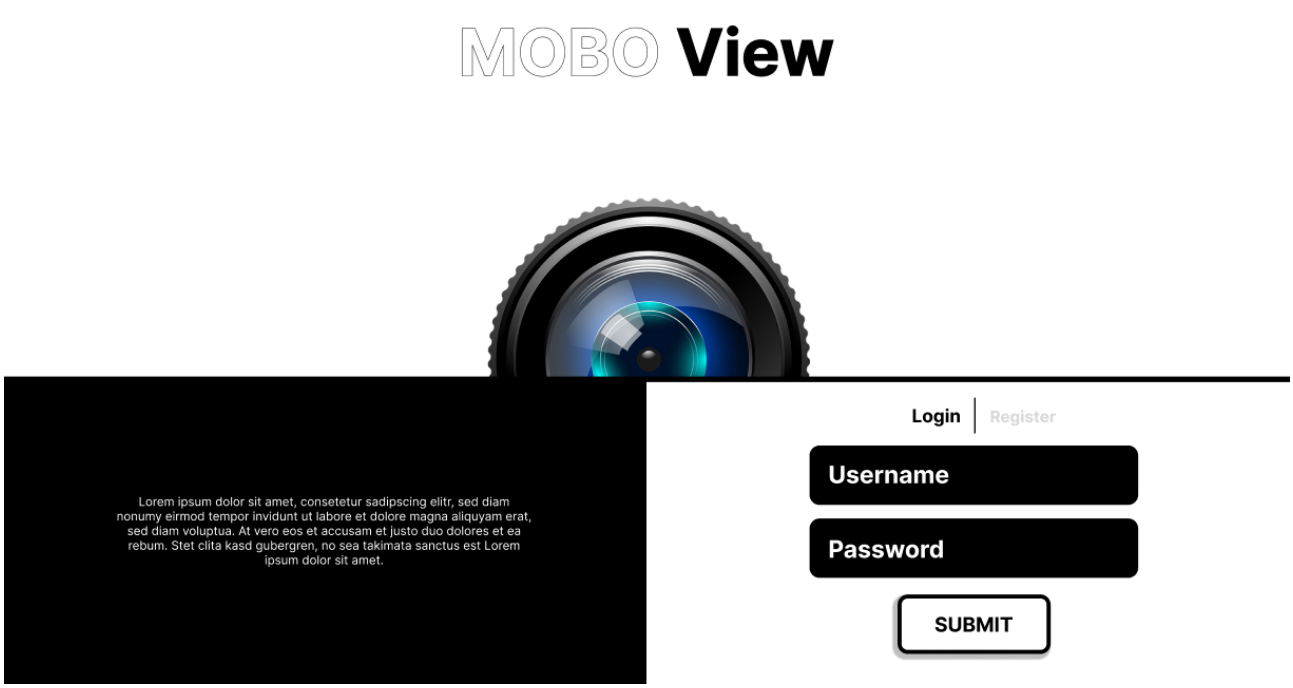


Abbildung 6: Landing Page

Ist man im System eingeloggt befindet sich auf der linken Seite eine globale Kameraliste (Siehe Abbildung 7) welche es je nach bedarf erlaubt Kamera-Elemente zu bearbeiten, zu löschen bzw. zum Dashboard hinzuzufügen.

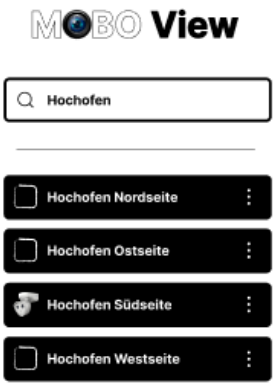


Abbildung 7: Kamera Liste

Nach Überlegungen wurde sich dafür entschieden ein Rollensystem zu implementieren.

Einerseits eine Rolle Viewer, mit welcher man ausschließlich dazu berechtigt ist, Kameras anzuzeigen und einen Admin-Request abzusetzen. Es dürfen somit als Viewer nur Kameras zum

Dashboard hinzugefügt und keine Editier- bzw. Löschoperationen vorgenommen werden (Siehe Abbildung 8). Auch das Registrieren neuer Kameras ist logischerweise nicht erlaubt.

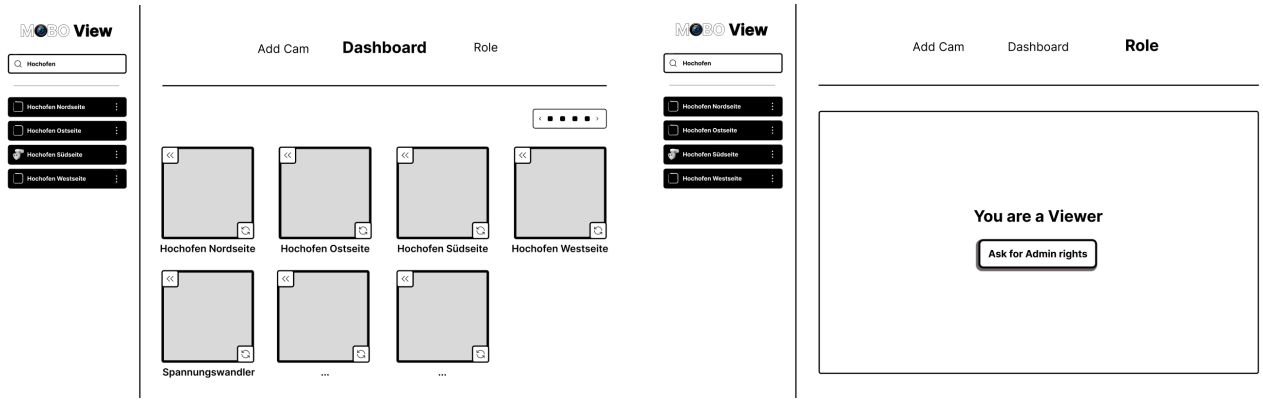


Abbildung 8: Viewer Ansicht

Dann gibt es auch noch eine Rolle Admin, welche es schließlich erlaubt Kameras zu editieren und diese aus der globalen Liste zu entfernen (Siehe Abbildung 10). Daraus lässt sich schlussfolgern, dass man als Admin ebenso berechtigt ist Kameras über eine Formulareingabe zu registrieren (Siehe Abbildung 9). Dies erfolgt durch die Eingabe einer Parameterkombination bestehend aus einer gültigen Kamera IP-Adresse und einer Bezeichnung.

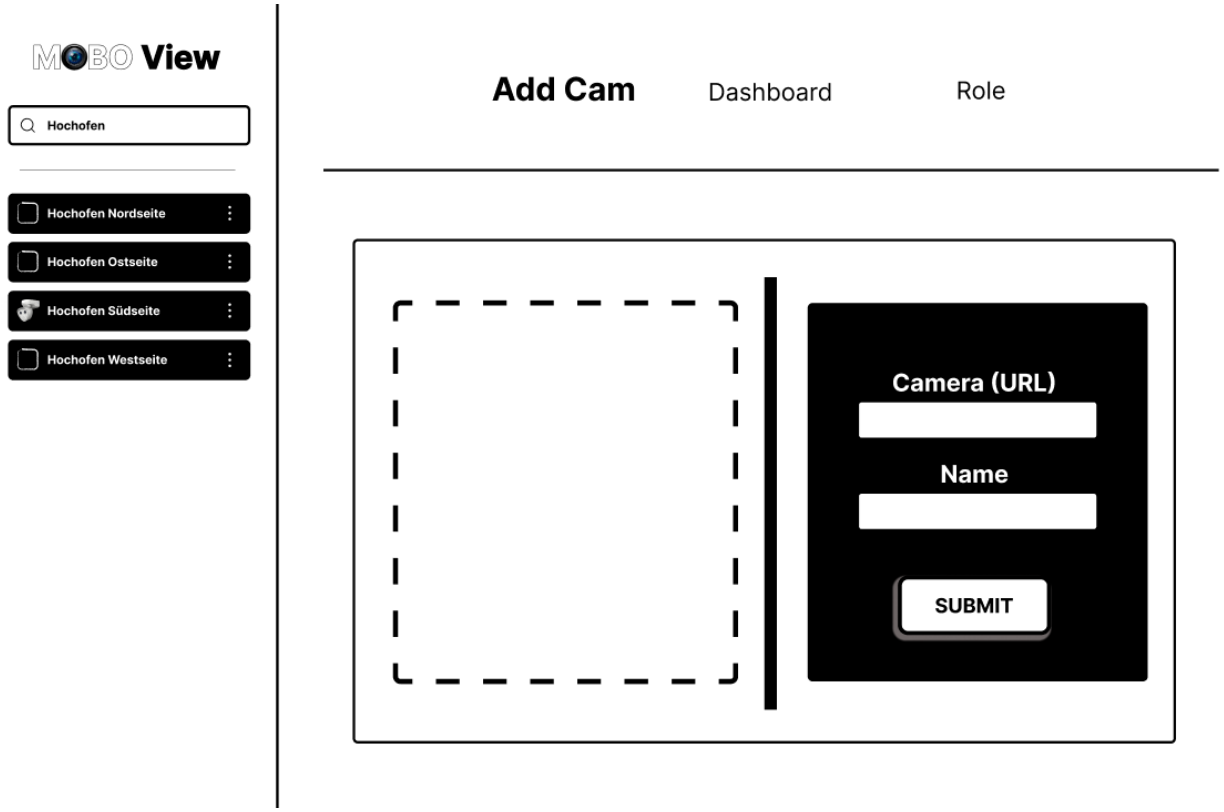


Abbildung 9: Add Camera

Die letzte der drei Rollen ist der sogenannte Super User. Diese Rolle wird im gesamten System nur einmalig vergeben. Der Super User ist nämlich ein Admin welcher zusätzlich die Rollen

Abbildung 10: Kameraelement Operationen als Admin

aller User bearbeiten und Admin-Requests akzeptieren bzw. ablehnen kann (Siehe Abbildung 11).

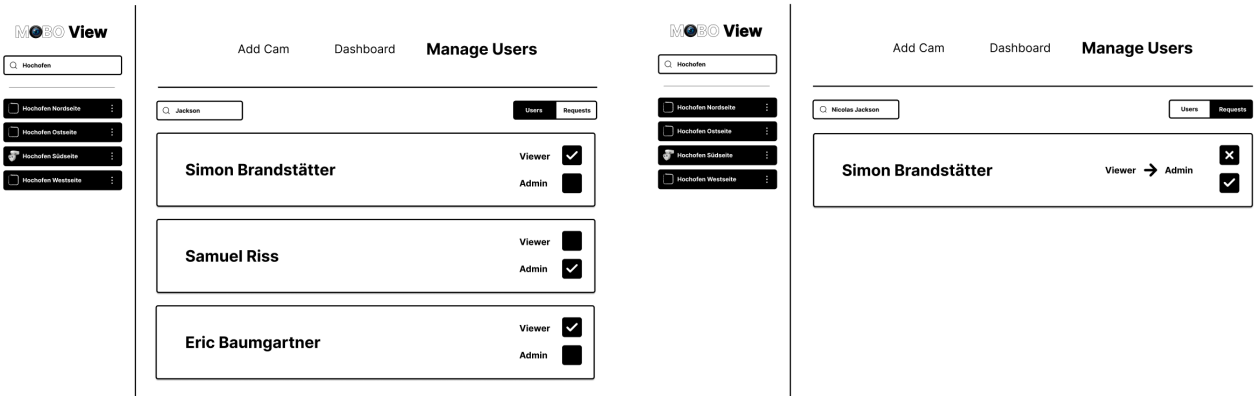


Abbildung 11: Super User Manage Users

Dann gibt es noch für alle Rollen die Möglichkeit, die Anzahl der Kameraelemete pro Reihe am Dashboard (Siehe Abbildung 12) nach eigenem Verlangen festzulegen.

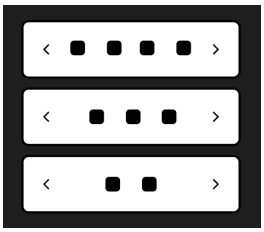


Abbildung 12: Anzahl der Kamera Elemente

2.1.2 Backend

2.1.3 Projektstruktur des Backends

Die Projektstruktur unterteilt sich in eine Backend-Schicht mit klar definierten Verantwortlichkeiten sowie eine Datenbankschicht mit Verbindungen und Initialisierungs-Skripten.

Root-Ebene

`app.ts` (siehe 4.3.1) ist der Einstiegspunkt der gesamten Anwendung, da dort der Express-Server initialisiert, die Middleware konfiguriert, Datenbankverbindungen hergestellt und alle

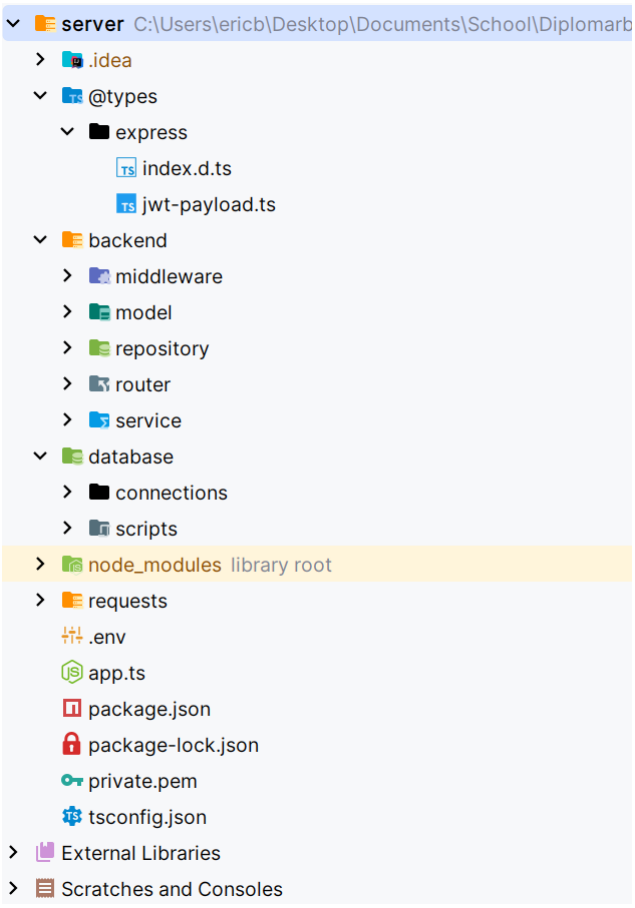


Abbildung 13: Projektstruktur des Backends

API-Router gemountet werden. Zu den Aufgaben von `app.ts` gehören das Erzeugen der Express-Anwendung, CORS-Konfiguration für sichere Cross-Origin-Anfragen (siehe 2.5.2), das Einbinden von Swagger/OpenAPI-Dokumentation (siehe 2.5.2 und 2.4.4), das Initialisieren von Datenbanken (siehe für Microsoft SQL Server 2.4.5 und für TimescaleDB 2.4.6), das Einrichten von geplanten Aufgaben (siehe 2.5.2) und das Montieren aller Router unter `/api`-Pfaden. `package.json` definiert währenddessen Abhängigkeiten, Entwicklungsabhängigkeiten und Start-Skripte wie `npm run backend` zum Starten des Servers im Watch-Modus.

/backend

Die Backend-Schicht ist in fünf Unterkategorien eingeteilt, die zusammenarbeiten, um HTTP-Anfragen zu verarbeiten.

Router: /backend/router

Unter `/router` liegen alle Router der Anwendung (siehe 4.3.4, 4.3.7, 4.3.5, 4.3.8 und 4.3.6). Ein Router definiert die HTTP-Endpunkte, also unter welcher URL etwas erreichbar ist, und HTTP-Methoden, also GET, POST, PUT und DELETE, einer API, wobei jeder Router für eine

spezifische Ressource verantwortlich ist, wie z.B. `user.router.ts` für die Benutzerverwaltung. Außerdem nutzen die Router für Authentifizierung und Autorisierung sogenannte Middleware, um die Sicherheit der Applikation zu verbessern. 2.4.3

Router	Funktionalität
user.router.ts	Benutzerverwaltung
camera.router.ts	Kameraverwaltung
temperature-data.router.ts	Temperaturmessdaten
role-request.router.ts	Rollenwechsel-Anfragen

Middleware: /backend/middleware

Middleware-Funktionen (siehe 4.3.3) verarbeiten HTTP-Anfragen bevor sie den Router erreichen, um zu prüfen, ob eine bestimmte Anfrage nach den aktuellen Umständen der Applikation oder eines bestimmten Users, erlaubt ist. Dadurch können Anfragen validiert, bearbeitet oder abgelehnt werden. Somit können Anfragen von beispielsweise uneingeloggten Usern oder jenen, die zu wenig Berechtigungen für eine bestimmte Aktion haben, abgewiesen werden. 2.4.3

Middleware	Funktion
auth.middleware.ts	JWT-Authentifizierung
role.middleware.ts	Rollenbasierte Zugriffskontrolle
validate-api-key.middleware.ts	API-Key-Validierung
validate-body.middleware.ts	Request-Body-Validierung

Model: /backend/model

Models sind TypeScript-Interfaces, die die Datenstrukturen der Anwendung definieren. Sie beschreiben, welche Felder eine Entität hat und welche Datentypen diese haben. Sie definieren z.B., dass sich ein User aus Benutzername, Passwort-Hash, Benutzererstellungsdatum und seiner Rolle zusammensetzt (siehe 7).

Listing 7: User Model

```
interface User {
    id: number;
    username: string;
    password_hash: string;
    created_at: Date;
    role: string;
}
```

Model	Beschreibung
user.model.ts	Benutzer
camera.model.ts	Kamera
session.model.ts	Session
temperature-data.model.ts	Temperaturmessung
user-camera.model.ts	Zuordnung zwischen Benutzern und Kameras
role-request.model.ts	Anfrage für Rollenänderung

Repository: /backend/repository

Das Repository-Pattern kapselt den Datenbankzugriff. Zu den typischen Aufgaben dieser Repositories gehört SQL-Abfragen ausführen, Datenbankfehler abfangen und loggen, Rohdaten in typisierte Models konvertieren und Datenbankverbindungen verwalten (siehe 4.3.4, 4.3.7, 4.3.5, 4.3.8 und 4.3.6).

Repository	Funktion
user.repository.ts	Benutzer-Datenbankoperationen
camera.repository.ts	Kamera-Datenbankoperationen
session.repository.ts	Session-Verwaltung
temperature-data.repository.ts	Temperaturmessdaten
role-request.repository.ts	Rollenwechsel-Anfragen

Service: /backend/service

Services (siehe 4.3.4 und 4.3.1) enthalten Geschäftslogik, die über einfache Datenbankzugriffe hinausgeht, wie das Hashen eines Passworts mit bcrypt (siehe 2.5.2) oder das Vergleichen eines gespeicherten Passwort-Hashes mit dem des eingegebenen Passworts.

Service	Funktion
auth.service.ts	Authentifizierungs-Logik
jwt-config.ts	JWT-Konfiguration
consistency.service.ts	Datenkonsistenz

Datenbankschicht: /database

Die Datenbankschicht ist in zwei Unterkategorien unterteilt. Die `connections`-Schicht ist für die Verbindung zu den Datenbanken verantwortlich, während die `scripts`-Schicht hauptsächlich zum Initialisieren der Datenbanken und Einfügen von Testdaten verwendet wird (siehe 2.1.5).

Connections: /database/connections

Dateiname	Datenbank
db-connection.mssql.ts	Microsoft SQL Server 2.4.5
db-connection.timescaledb.ts	PostgreSQL 2.4.5 mit TimescaleDB 2.4.6-Extension 2.4.6

Scripts: /database/scripts

Skript	Funktion
mssql-setup.sql	MSSQL-Datenbankschema
timescaledb-setup.sql	TimescaleDB-Schema und Hypertable-Konfiguration
setup-superuser.sql	Erstellt einen Initial-Superuser
testdata.sql	Beispieldaten für Entwicklung und Tests
insert-temperature-testdata.sql	Einfügen von Beispiel-Temperaturmessdaten
runMSSqlScript.ts	TypeScript-Utility zum Ausführen von MSSQL-Skripten
runTimescaleSqlScript.ts	TypeScript-Utility zum Ausführen von TimescaleDB-Skripten

TypeScript-Definitionen: /@types

Das Verzeichnis `@types/` enthält TypeScript-Definitionen für Express.js (siehe 2.4.3). Die Datei `@types/express/index.d.ts` definiert die Typen für Request- und Response-Objekte, während `@types/express/jwt-payload.ts` die Struktur der JWT-Payload (siehe 2.4.5) beschreibt, also der Daten, die im Token, einer verschlüsselten Zeichenkette zur Authentifizierung, enthalten sind. Der Server sendet diesen Token nach erfolgreichem Login an den Client, der ihn bei weiteren Anfragen mitsendet. Die Middleware extrahiert die Benutzerinformationen daraus und speichert sie in `req.user`, ein Objekt mit ID, Username und Rolle. Diese Definitionen ermöglichen IDE-Unterstützung und Type-Checking, insbesondere bei der Nutzung von `req.user` in Routern und Middleware. Dadurch werden Typfehler bereits beim Schreiben des Codes erkannt, statt später zur Laufzeit bei einem Fehlschlag.

API-Testing/-Dokumentation: /requests

`api-tests.http` enthält eine Sammlung von HTTP-Test-Requests im `restclient`-Format, was das direkte Testen der API in der IDE ermöglicht. Somit kann getestet werden, ob Login, CRUD-Operationen, usw. korrekt funktionieren.

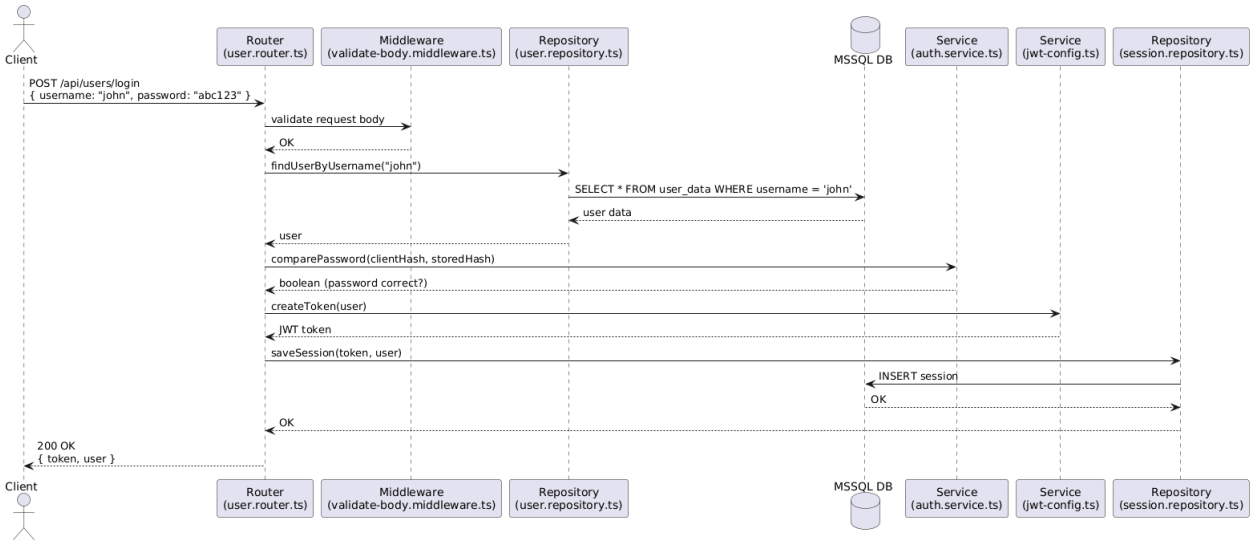


Abbildung 14: Ablauf eines User-Logins mit Abbildung der Kommunikation aller Komponenten

2.1.4 Rollenmodell

Rollen im System

Das Rollenmodell besteht aus drei verschiedenen Rollen, die in einer Hierarchie angeordnet sind.

Viewer (Standard-Benutzer)

`Viewer` ist die Standard-Rolle, die neue Benutzer bei der Registrierung erhalten. `Viewer` haben die niedrigsten Berechtigungen, weshalb sie nur ihre eigenen Benutzerdaten abrufen, Verlinkungen auf bereits vorhandene Kameras für sich selbst erstellen, sowie Temperaturmessdaten dieser einsehen können. Sie können auch Anfragen zur Beförderung an den `Superuser` stellen, um erweiterte Berechtigungen zu erhalten. `Viewer` dürfen jedoch keine bestehenden Kameras im System löschen oder erstellen.

Admin

Die `Admin`-Rolle ermöglicht erweiterte administrative Funktionen. `Admins` können zusätzlich zu allen Berechtigungen des `Viewers` Kameras in das System aufnehmen und bestehende Kameras

löschen. Sie haben Zugriff auf alle Kameras des Systems, können aber keine anderen Benutzer verwalten oder über Beförderungsanfragen dieser entscheiden. Diese Rolle wird nicht automatisch vergeben, sondern muss beantragt und von einem `Superuser` gewährt werden.

Superuser

Der `Superuser` ist die höchste Rolle mit uneingeschränkten Rechten. Er kann alle Benutzer des Systems verwalten, alle Beförderungsanfragen einsehen und genehmigen oder ablehnen, sowie auf alle Kameras zugreifen. Es existiert nur ein `Superuser` im System, der beim Datenbankboot als `'owner'` erstellt wird und die zentrale administrative Verwaltung übernimmt.

Rollenänderungsprozess

Als `Viewer` ist keine eigenständige Rollenänderung durch den Benutzer möglich, weshalb ein kontrollierter Ablauf zur Rechtevergabe entwickelt wurde. Somit können User eine Anfrage auf Beförderung zur `Admin`-Rolle stellen, welche dann zunächst mit dem Status `'pending'` gespeichert wird. Der `Superuser` sieht alle offenen Anträge in einer Übersicht im Frontend, um sie in weiterer Folge genehmigen (`'accepted'`) oder ablehnen (`'declined'`) zu können. Sollte ein User des Öfteren Probleme bereiten, z.B. durch das Löschen wichtiger Kameras aus dem System, kann der `Superuser` ihn auch wieder zum `Viewer` degradieren. Dieser Rollenänderungsprozess verhindert somit eine unkontrollierte Rechteausweitung und Selbstbeförderung nicht befugter User.

Sicherheitsaspekte

Das Backend implementiert ein mehrstufiges Sicherheitskonzept mit klarer Trennung der Rollen und Zuständigkeiten, wodurch kritische Funktionen wie die Benutzerverwaltung ausschließlich dem Superuser zur Verfügung stehen, um wirksam vor 'Privilege-Escalation' [4], dem unkontrollierten Befördern der User zum `Admin`, zu schützen. Zur zusätzlichen Sicherheit werden die Benutzerrechte bei jeder Anfrage erneut überprüft, um eine laufende Autorisierung zu gewährleisten. Selbst im Fall gestohlener Tokens bleibt der Zugriff auf die ursprünglichen Rechte des Benutzers beschränkt. Da die hierarchische Struktur dem sogenannten 'Least Privilege Principle' [5] folgt, erhält jeder Benutzer nur die Rechte, die er für seine Aufgaben benötigt, um das Risiko, dass ein kompromittiertes Benutzerkonto mit sinnlosem Zugriff auf sensible Operationen immensen Schaden anrichten kann, so gut wie möglich zu reduzieren.

2.1.5 Aufbau der Datenbanken

Datenbankstruktur

Das System im Backend folgt einer hybriden Datenbank-Strategie, die eine relationale Datenbank (MSSQL 2.4.5) mit einer Zeitreihendatenbank (TimescaleDB 2.4.6) kombiniert. Diese Aufteilung ergibt sich durch die unterschiedlichen Anforderungen an die beiden Teilsysteme. Während MSSQL transaktionale Daten wie Benutzer, Sessions, Kameras und Zugriffskontrolle (siehe 2.1.4) verwaltet und dabei ACID-Eigenschaften sowie Datenkonsistenz gewährleistet, ist TimescaleDB für die effiziente Speicherung und Abfrage großer Mengen an Zeitreihendaten, bzw. Temperaturmessdaten der Kameras, optimiert. Diese Spezialisierung ermöglicht eine optimale Ressourcennutzung für unterschiedliche Workloads und erfüllt damit die Anforderungen an das System nach hoher Verfügbarkeit, Zuverlässigkeit und schnellen Abfragen auch bei großen Datenmengen, während gleichzeitig die Konsistenz und Datensicherheit durch Passwort-Hashing (siehe 2.5.2 sowie durch die Verschlüsselung der Kamerazugangsdaten 106 gewährleistet wird. (siehe 4.3)

Für die verschiedenen Datentypen, Constraints und Indizes in MSSQL sowie TimescaleDB, welche auf PostgreSQL (siehe 2.4.5) basiert, wurde sich auf folgende Dokumentationen bezogen: [6] [7] [8] [9] [10] [11] [12] [13]

Ein Index ist eine Datenstruktur, die Abfragen beschleunigt, indem die Datenbank gezielt zur gesuchten Reihe springen kann, statt alle Reihen durchsuchen zu müssen. Ein Constraint ist eine Regel, die die Datenqualität sicherstellt, indem es bestimmte Bedingungen erzwingt (z.B. `UNIQUE, NOT NULL, FOREIGN KEY`). `IF NOT EXISTS` wurde fast überall verwendet, um keine Probleme beim Starten/Stoppen des Backends zu bekommen und nicht immer alles neu erstellen/droppen zu müssen.

MSSQL-Datenbank 'devdb'

user_data speichert Benutzerinformationen und Authentifizierungsdaten.

Attribut	Definition	Funktion
`id`	INT, PK, IDENTITY	Eindeutige Benutzer-ID
`username`	VARCHAR, UNIQUE, NOT NULL	Eindeutiger Benutzername
`password_hash`	TEXT, NOT NULL	Gehashtes Passwort des Users
`created_at`	DATETIMEOFFSET, NOT NULL	Zeitstempel der Kontoerstellung
`role`	VARCHAR, NOT NULL, `'viewer'`	Rolle des Benutzers (siehe 2.1.4)

In `user_data` gibt es einen Primärschlüssel-Index auf `id` und zusätzlich einen eindeutigen Index auf `username`, da SQL Server für die Constraints `PRIMARY KEY` und `UNIQUE` automatisch jeweils einen eindeutigen Index erstellt, um Eindeutigkeit effizient zu prüfen und schnelle Suchabfragen zu ermöglichen. Wenn ein User erstellt wird, wird per `DEFAULT` seine Rolle auf `'viewer'` gesetzt. `IDENTITY` sorgt dafür, dass das Feld `id` mit jedem neuen Eintrag inkrementiert wird, damit jeder User eine eindeutige ID hat. Der Check-Constraint bei `role` stellt sicher, dass diese nur die Werte `'viewer'`, `'admin'` oder `'super_user'` annehmen kann.

sessions verwaltet authentifizierte Benutzer-Sessions und JWT-Tokens (siehe 2.4.5, 4.3.4 und 4.3.5).

Attribut	Definition	Funktion
`id`	INT, PK, IDENTITY	Eindeutige Session-ID
`token`	VARCHAR, NOT NULL, UNIQUE	JWT-Token des Users
`user_id`	INT, NOT NULL, FK	Referenz zu `user_data(id)`
`expires_at`	DATETIMEOFFSET, NOT NULL	Ablaufzeit des Tokens
`refresh_token`	VARCHAR, NOT NULL	Token für Session-Erneuerung

Die Tabelle `sessions` verwaltet authentifizierte Benutzer-Sessions und speichert JWT-Tokens für API-Anfragen. Jede Session wird durch eine eindeutige `id` identifiziert und referenziert einen Benutzer über `user_id` mit `ON DELETE CASCADE`, wodurch Sessions beim Benutzer-Löschen automatisch entfernt werden. Der sogenannte 'Access Token' (`token`) wird mit einem eindeutigen Index gespeichert, um schnelle Datenabfragen zu ermöglichen und Duplikate zu verhindern. Der `refresh_token` erlaubt die Erneuerung von Sessions, bzw. des Access Tokens, ohne erneute Anmeldung und wird über einen zusätzlichen nicht-eindeutigen Index unterstützt. Das Feld `expires_at` definiert, bis zu welchem Zeitpunkt die Session gültig ist. Sicherheit wird durch die CASCADE-Delete-Regel und eine regelmäßige Bereinigung abgelaufener Tokens durch einen Cron-Job (siehe 2.5.2 und 4.3.1) gewährleistet.

Der `token` (Access Token) ist durch einen `UNIQUE INDEX` eindeutig, da er bei jedem API-Request validiert werden muss und eine schnelle eindeutige Zuordnung zur Session erfordert, während der `refresh_token nicht UNIQUE` sein kann, da die kryptographische Token-Generierung (siehe 2.4.5 und 2.5.2) Duplikate praktisch ausschließt, wodurch `UNIQUE` bei Tokens optional ist. Die Sicherheit wird in beiden Fällen durch die Verknüpfung zwischen `token` und `user_id` garantiert. Somit kann ein User mit `id = 5` keine Anfragen mit dem Access Token von dem User mit `id =`

7 versenden, da die Verknüpfung das verhindert. Dadurch, dass `user_id` nicht einzigartig ist, kann ein Benutzer auch auf mehreren Geräten gleichzeitig eingeloggt sein.

cameras speichert Kameras und deren Verbindungsinformationen.

Attribut	Definition	Funktion
`id`	INT, PK, IDENTITY	Eindeutige Kamera-ID
`camera_name`	VARCHAR, NOT NULL, UNIQUE	Name der Kamera
`stream_url`	VARCHAR, NOT NULL, UNIQUE	Video-Stream-URL der Kamera
`camera_username`	VARCHAR, NOT NULL	Kamera-Benutzername
`camera_pw_encrypted`	VARCHAR, NOT NULL	Verschlüsseltes Kamera-Passwort

`stream_url` ist die HTTP Stream-URL, während `camera_username` und `camera_pw_encrypted` die Zugangsdaten der Kamera sind. Sie werden benötigt, damit sich die Anwendung mit der Kamera verbinden kann. Automatisch erstellte Indizes auf den Primary Key und Attribute mit `UNIQUE`-Constraints sind ebenfalls wieder vorhanden.

user_cameras ist die Many-to-Many-Zuordnungstabelle zwischen Benutzern und Kameras, um speichern zu können, welcher User mit welcher Kamera verlinkt ist.

Attribut	Definition	Funktion
`user_id`	INT, NOT NULL, FK	Referenz zu `user_data(id)`
`camera_id`	INT, NOT NULL, FK	Referenz zu `cameras(id)`
`camera_lens`	VARCHAR, NOT NULL, 'normal'	Ansichtsmodus
`is_alarm_active`	BIT, NOT NULL, 1	Alarmbenachrichtigung

Unterschied zwischen einer echten Kamera im System und nur einer Verlinkung eines Users auf eine: `cameras` enthält die echten, physischen Kameras im System, also den eigentlichen Datensatz mit Name, Stream-URL und Zugangsdaten, während `user_cameras` nur die Zuordnungstabelle ist, die festlegt, welcher Benutzer auf welche Kamera verweist. Das ist besonders wichtig für die Ansicht im Frontend, damit der User nicht alle existierenden Kameras angezeigt bekommt, sondern nur die, auf die er verweist. Somit hat er eine Übersicht mit allen für ihn relevanten Kameras. Außerdem kann dann noch individuell für jede verlinkte Kamera eingestellt werden, welche Ansicht angezeigt und ob man bei einem Alarm benachrichtigt werden soll. Somit kann jeder Benutzer einstellen, welche Kameras er wie angezeigt bekommen möchte, ohne dass die eigentlichen Kameras im System oder die Einstellungen anderer User

dadurch beeinflusst werden. Eine „echte“ Kamera ist also ein eigener Eintrag in `cameras`, eine „Verlinkung“ ist nur ein Verweis darauf in `user_cameras` und erzeugt keine neue Kamera, sondern nur eine Beziehung für einen Nutzer.

`user_id` und `camera_id` haben den Foreign Key Constraint `ON DELETE CASCADE`, sodass beim Löschen eines Users oder einer Kamera auch alle dazugehörenden Einträge in `user_cameras` automatisch gelöscht werden, um verwaiste Einträge in dieser Tabelle zu vermeiden. `camera_lens` ist standardmäßig auf `'normal'`, jedoch kann durch das Drücken eines Buttons im Frontend diese Ansicht auf `'thermal'` umgestellt werden, um das Wärmebild der Kamera zu sehen. Mit demselben Knopf kann auch wieder auf `'normal'` zurückgewechselt werden. `is_alarm_active` ist standardmäßig `'true'`, wodurch man bei einem Alarm in der Applikation im Frontend benachrichtigt wird, jedoch kann diese Alarmbenachrichtigung auch jederzeit deaktiviert bzw. reaktiviert werden. Der Constraint `PRIMARY KEY (user_id, camera_id)` verhindert doppelte Einträge in der Tabelle und sichert die referenzielle Integrität der Daten. Automatisch erstellte Indizes durch z.B. `PRIMARY KEY` sind ebenfalls wieder vorhanden.

role_requests wurde für den Genehmigungsworkflow von Rollenänderungen erstellt (siehe 2.1.4).

Attribut	Definition	Funktion
`id`	INT, PK, IDENTITY	Eindeutige Request-ID
`user_id`	INT, NOT NULL, FK	Referenz zu `user_data(id)`
`requested_role`	VARCHAR, NOT NULL, 'admin'	Gewünschte Rolle
`status`	VARCHAR, NOT NULL, 'pending'	Status des Requests
`created_at`	DATETIMEOFFSET, NOT NULL	Timestamp der Anfrage
`updated_at`	DATETIMEOFFSET, NOT NULL	Timestamp der letzten Änderung

Die Tabelle implementiert einen Genehmigungsworkflow für Rollenerhöhungen von `viewer` zu `admin`. Sie enthält die `id` als Primary Key mit Identity, `user_id` als Foreign Key mit `ON DELETE CASCADE`, `requested_role` für die gewünschte Rolle, wobei nur `'admin'` möglich ist, und das zentrale Statusfeld mit den erlaubten Werten `'pending'`, `'accepted'` oder `'declined'`. Die Felder `created_at` und `updated_at` speichern Zeitstempel, wobei `updated_at` bei Statusänderungen wie von `'pending'` zu `'accepted'` oder `'declined'` aktualisiert wird.

Workflow: Ein `viewer` kann eine Beförderungsanfrage zu `admin` stellen, die mit Status `'pending'` in der Datenbank gespeichert wird. Der `super_user` kann diese dann akzeptieren oder ablehnen. Der gefilterte Index `IX_role_requests_user_pending` mit `WHERE status`

= 'pending' als UNIQUE Index verhindert mehrere ausstehende Anfragen pro Benutzer. Ein weiterer Index IX_role_requests_status ermöglicht schnelle Abfragen aller ausstehenden Anfragen. Wird ein 'admin' später zum 'viewer' degradiert, wird sein akzeptierter Request automatisch auf 'declined' gesetzt. Somit existiert immer nur ein Role-Request pro User, welcher bei Änderungen aktualisiert wird.

Listing 8: MSSQL-Initialisierungsskript

```
-- This script sets up the MSSQL database and tables for the application.
-- It assumes a fresh DB may be created on each server/container start, but is now idempotent so it can be executed multiple times safely.

IF DB_ID('devdb') IS NULL
BEGIN
    CREATE DATABASE devdb;
END
GO

USE devdb;
GO

-- Users
IF OBJECT_ID('user_data', 'U') IS NULL
BEGIN
    CREATE TABLE user_data (
        id INT IDENTITY PRIMARY KEY,
        username VARCHAR(100) UNIQUE NOT NULL,
        password_hash TEXT NOT NULL,
        created_at DATETIMEOFFSET NOT NULL DEFAULT SYSDATETIMEOFFSET(),
        role VARCHAR(50) NOT NULL DEFAULT 'viewer' CONSTRAINT CK_user_data_role CHECK (role IN ('viewer','admin','super_user'))
    );
END
GO

-- Sessions
IF OBJECT_ID('sessions', 'U') IS NULL
BEGIN
    CREATE TABLE sessions (
        id INT IDENTITY PRIMARY KEY,
        token VARCHAR(2000) NOT NULL,
        user_id INT NOT NULL FOREIGN KEY REFERENCES user_data(id) ON DELETE CASCADE,
        expires_at DATETIMEOFFSET NOT NULL,
        refresh_token VARCHAR(1000) NOT NULL
    );
END
GO

-- Index to speed lookup by token and refresh token
IF NOT EXISTS (
    SELECT 1 FROM sys.indexes WHERE name = 'IX_sessions_token' AND object_id = OBJECT_ID('sessions')
)
BEGIN
    CREATE UNIQUE INDEX IX_sessions_token ON sessions(token);
END
GO

IF NOT EXISTS (
    SELECT 1 FROM sys.indexes WHERE name = 'IX_sessions_refresh' AND object_id = OBJECT_ID('sessions')
)
BEGIN
    CREATE INDEX IX_sessions_refresh ON sessions(refresh_token);
END
GO

-- Cameras
IF OBJECT_ID('cameras', 'U') IS NULL
BEGIN
    CREATE TABLE cameras (
        id INT IDENTITY PRIMARY KEY,
        camera_name VARCHAR(255) UNIQUE NOT NULL,
        stream_url VARCHAR(1000) UNIQUE NOT NULL,
        camera_username VARCHAR(255) NOT NULL,
```

```
64          camera_pw_encrypted VARCHAR(1000) NOT NULL
65      );
66  END
67  GO
68
69  -- User-Camera link
70  IF OBJECT_ID('user_cameras', 'U') IS NULL
71  BEGIN
72      CREATE TABLE user_cameras (
73          user_id INT NOT NULL REFERENCES user_data(id) ON DELETE CASCADE,
74          camera_id INT NOT NULL REFERENCES cameras(id) ON DELETE CASCADE,
75          camera_lens VARCHAR(50) NOT NULL DEFAULT 'normal',
76          is_alarm_active BIT NOT NULL DEFAULT 1,
77          PRIMARY KEY (user_id, camera_id)
78      );
79  END
80  GO
81
82  -- Role requests
83  IF OBJECT_ID('role_requests', 'U') IS NULL
84  BEGIN
85      CREATE TABLE role_requests (
86          id INT IDENTITY PRIMARY KEY,
87          user_id INT NOT NULL REFERENCES user_data(id) ON DELETE CASCADE,
88          requested_role VARCHAR(50) NOT NULL,
89          status VARCHAR(20) NOT NULL DEFAULT 'pending' CONSTRAINT CK_role_requests_status
                CHECK (status IN ('pending','accepted','declined')),
90          created_at DATETIMEOFFSET NOT NULL DEFAULT SYSDATETIMEOFFSET(),
91          updated_at DATETIMEOFFSET NOT NULL DEFAULT SYSDATETIMEOFFSET()
92      );
93  END
94  GO
95
96  -- Ensure max 1 pending request per user (filtered unique index)
97  IF NOT EXISTS (
98      SELECT 1 FROM sys.indexes WHERE name = 'IX_role_requests_user_pending' AND object_id =
            OBJECT_ID('role_requests')
99  )
100 BEGIN
101     CREATE UNIQUE INDEX IX_role_requests_user_pending ON role_requests(user_id) WHERE
            status = 'pending';
102 END
103 GO
104
105 -- Optional index to speed up status queries
106 IF NOT EXISTS (
107     SELECT 1 FROM sys.indexes WHERE name = 'IX_role_requests_status' AND object_id =
            OBJECT_ID('role_requests')
108 )
109 BEGIN
110     CREATE INDEX IX_role_requests_status ON role_requests(status);
111 END
112 GO
113
114 -- End of script
```

TimescaleDB - Zeitreihendatenbank

TimescaleDB wird für die Speicherung und schnelle Abfrage großer Mengen an Zeitreihendaten verwendet, insbesondere für Temperaturmessdaten der Wärmebildkameras. Die Datenbank ist spezialisiert auf append-only Workloads und optimiert für Zeitreihendaten (siehe 2.4.6). Dafür nutzt TimescaleDB das Konzept der Hypertables, die Daten automatisch nach Zeitstempel partitionieren bzw. in Chunks aufteilen, was zu einer deutlich effizienteren Speichernutzung und schnellen Abfragen durch Kompression alter Chunks führt. Die Tabelle `temperature_data` wird als Hypertable auf dem Zeitstempel-Feld `time` erstellt und speichert neben der gemessenen Temperatur auch die `camera_id`, einen Konfidenzwert `confidence` zwischen 0 und 1, welcher aussagt, inwiefern sich die KI-Komponente sicher ist, ob der analysierte Temperaturwert auf einen Brand

hindeutet, sowie ein `is_risky`-Flag, das anzeigt, ob das Ergebnis kritisch ist. Es sind zwei spezialisierte Indizes vorhanden. `idx_temperature_data_time_desc` auf dem `time`-Feld in absteigender Reihenfolge für schnelle Einzelabfragen und `idx_temperature_data_camera_time_desc` als zusammengesetzter Index auf `camera_id` und `time`, um zeitliche Bereiche pro Kamera effizient abzurufen.

temperature_data speichert Temperaturdaten und die Branderkennung pro Kamera.

Attribut	Definition	Funktion
`time`	TIMESTAMPTZ, NOT NULL, PK	Messungszeitstempel
`temperature`	DOUBLE PRECISION, NOT NULL	Gemessene Temperatur
`camera_id`	BIGINT, NOT NULL, PK	Referenz `cameras(id)`
`confidence`	DOUBLE PRECISION, NOT NULL	Konfidenzwert des KI-Modells
`is_risky`	BOOLEAN, NOT NULL, DEFAULT false	Temp. kritisch/alarmierend?

Listing 9: TimescaleDB-Initialisierungsskript

```
1  -- TimescaleDB Setup Script
2  -- This script initializes the TimescaleDB database for temperature data.
3
4  -- Ensure TimescaleDB extension is available
5  CREATE EXTENSION IF NOT EXISTS timescaledb;
6
7  CREATE TABLE IF NOT EXISTS temperature_data (
8      time        TIMESTAMPTZ       NOT NULL,
9      temperature DOUBLE PRECISION  NOT NULL,
10     camera_id   BIGINT            NOT NULL,
11     confidence  DOUBLE PRECISION  NOT NULL,
12     is_risky    BOOLEAN           NOT NULL DEFAULT FALSE,
13     CONSTRAINT chk_confidence_range CHECK (confidence >= 0 AND confidence <= 1),
14     PRIMARY KEY (time, camera_id)
15 );
16
17 -- Create TimescaleDB hypertable for efficient time-series storage
18 SELECT create_hypertable('temperature_data', 'time', if_not_exists => TRUE);
19
20 -- Helpful indexes for common query patterns
21 CREATE INDEX IF NOT EXISTS idx_temperature_data_time_desc ON temperature_data (time DESC);
22 CREATE INDEX IF NOT EXISTS idx_temperature_data_camera_time_desc ON temperature_data
       (camera_id, time DESC);
```

Cross-Database-Beziehungen

Die referenzielle Integrität zwischen den beiden Datenbanken wird durch das logische Konzept realisiert, dass `temperature_data.camera_id` auf `cameras.id` in MSSQL verweist. Allerdings können klassische Fremdschlüssel nicht zwischen verschiedenen Datenbanksystemen definiert werden, weshalb die Validierung auf Applikationsebene erfolgt. Das bedeutet, dass die API vor dem Speichern von Temperaturdaten in TimescaleDB prüft, ob die `camera_id` in MSSQL

existiert. Es werden auch regelmäßig Clean-Up-Jobs ausgeführt (siehe 2.5.2), welche verwaiste Temperaturdaten nach dem Löschen einer Kamera ebenfalls löschen.

Entity Relation Diagram

Um den Zusammenhang zwischen allen Entitäten und Tables noch besser zu veranschaulichen, wurde zusätzlich ein ERD entworfen.

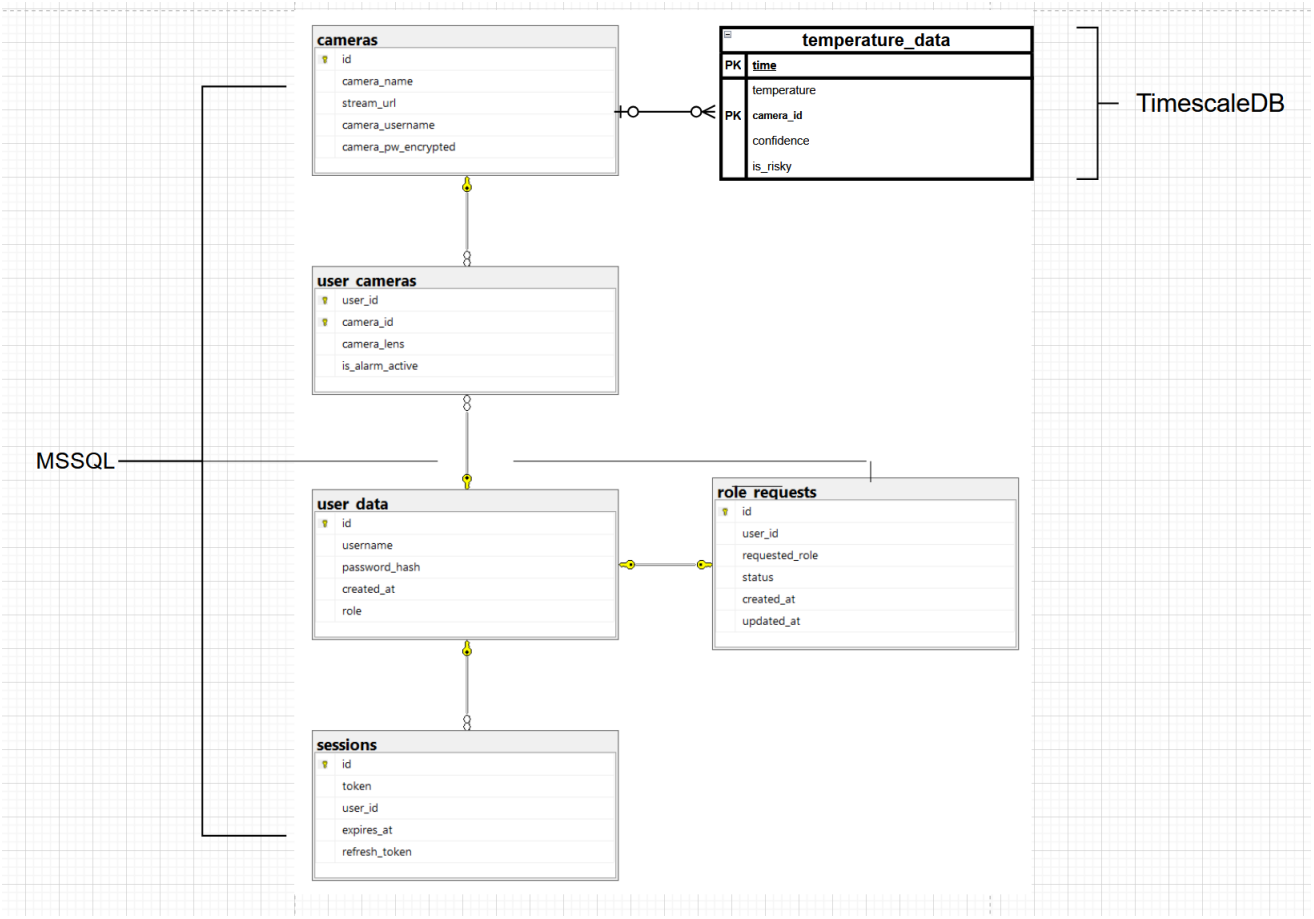


Abbildung 15: ERD des gesamten Datenmodells

Kardinalitäten

- `user_data (1) : sessions (N)`
- `user_data (1) : user_cameras (N)`
- `cameras (1) : user_cameras (N)`
- `user_data (1) : role_requests (N)`
- `cameras (1) : temperature_data (N)`

2.1.6 Umgebungskonfiguration und Sicherheit

Umgebungskonfiguration - .env

Die Konfiguration der Applikation erfolgt über die `.env`-Datei, welche alle sensiblen Informationen wie Datenbankverbindungen, API-Schlüssel und kryptografische Secrets zentralisiert speichert. Diese Datei wird später nach der Inbetriebnahme beim Auftraggeber zum Schutz der Anmeldeinformationen nicht mehr in ein Git-Repository committed und muss sicher am Produktionsserver aufbewahrt werden. Die `.env`-Datei enthält die Verbindungsdetails für beide Datenbanksysteme (MSSQL und TimescaleDB(siehe 2.4.5 und 2.4.6)) inklusive Benutzername, Passwort, Servername und Port. Zusätzlich werden dort der `JWT_SECRET` für die Token-Signierung mit dem HS256-Algorithmus und der `BCRYPT_ROUNDS`-Parameter gespeichert (siehe 2.4.5 und 2.5.2). Die API-Schlüssel für interne Services wie den Flask-Server sowie Flags wie `CONSISTENCY_AUTOFIX`, welche automatische Cleanups verwaister Temperaturdaten aktiviert, und `SWAGGER_ENABLED`, was die API-Dokumentation (siehe 2.4.4 und 2.5.2) aktiviert bzw. deaktiviert, sind ebenfalls in der `.env`-Datei konfigurierbar. Durch das saubere Trennen von Konfiguration und Code wird sichergestellt, dass unterschiedliche Umgebungen, wie Entwicklung oder Produktion, mit unterschiedlichen Secrets bzw. .env-Files betrieben werden können, ohne dass vertrauliche Informationen im Quellcode offengelegt werden.

Verschlüsselung von Kameraanmeldeinformationen - private.pem

Für die Verschlüsselung von sensiblen Kamerainformationen, insbesondere der Kamera-Zugangsdaten, wird ein RSA-Private-Key verwendet, der in der `private.pem`-Datei im PEM-Format gespeichert ist. Dieser Schlüssel wird beim Starten der Applikation durch die Funktion `decryptCameraPassword()` aus dem Dateisystem geladen und dient dazu, verschlüsselte Kamera-Passwörter mit dem RSA-OAEP-Padding-Schema zu entschlüsseln (siehe 2.4.5). Die `private.pem`-Datei wird, ähnlich wie die `.env`-Datei später in der Produktion, nicht ins Git-Repository committed und muss auf dem Produktionsserver sicher aufbewahrt werden. Dadurch wird sichergestellt, dass Kamera-Zugangsdaten, auch wenn die Datenbank kompromittiert wird, nicht im Klartext verfügbar sind, sondern nur mit dem Private-Key entschlüsselbar bleiben.

2.2 Fachbegriffe

2.2.1 Data Augmentation

Beim Training von ML-Modellen treten häufig Herausforderungen im Zusammenhang mit den verfügbaren Trainingsdaten auf. Probleme entstehen insbesondere dann, wenn geeignete Datensätze für ein Projekt erst erzeugt werden müssen. Für eine gute Generalisierungsfähigkeit sind in der Regel große Datenmengen erforderlich. In der Praxis ist es jedoch häufig nicht möglich, diese vollständig aus realen Szenarien zu gewinnen, da dies mit hohem zeitlichem Aufwand und erheblichen Kosten verbunden ist.

Ein weiteres Problem stellen unausgewogene Datensätze dar, bei denen beispielsweise viele Daten für einen Fall vorhanden sind, während andere Fälle nur unzureichend repräsentiert sind.

An dieser Stelle kommt die Data Augmentation zum Einsatz. Durch gezielte und kontrollierte Modifikation vorhandener Daten wird künstlich zusätzliche Varianz erzeugt, wodurch neue Trainingsdaten auf Basis der bestehenden Datensätze generiert werden. Hierfür existieren verschiedene Verfahren. Bei Bilddaten werden beispielsweise geometrische Transformationen wie Rotation oder Skalierung sowie photometrische Transformationen wie die Variation von Helligkeit und Kontrast angewendet. [14]

2.2.2 Thread

Ein Thread stellt eine eigene Ausführungseinheit innerhalb eines Prozesses dar. Mithilfe von Threads können multiple Instanzen derselben Funktion ausgeführt werden. Bei Client-Server-Architekturen wird diese Technologie verwendet, um am Server mehrere Anfragen parallel bearbeiten zu können. Weitere Informationen über Threads in Python sind in Abschnitt 2.4.8 zu finden.

2.2.3 Neuronales Netzwerk

Zur Klassifikation von Daten (siehe Abschnitt 2.2.4) werden oft neuronale Netze eingesetzt. Ein neuronales Netzwerk kann den Zusammenhang zwischen Features (siehe Abschnitt 2.2.5) und einer Zielklasse lernen, ohne konkrete Hinweise zu diesem Zusammenhang zu haben. Möglich gemacht wird dies durch Supervised Learning (siehe Abschnitt 2.2.6).
Ein neuronales Netzwerk besteht aus drei Arten von Schichten:

- Input-Layer

  Diese Schicht ist dafür zuständig, die Features der Eingabedaten an die Hidden Layer weiterzugeben. Sie gibt auch die Form der Eingabe vor. Hier findet noch keine Verarbeitung der Daten statt.
- Hidden Layer

  Ein Modell hat multiple Hidden Layer. Um so tiefer ein Hidden Layer ist, um so komplexer sind die möglichen Zusammenhänge die diese Schicht abbilden kann. Jede dieser Schichten erhält die Ergebnisse der vorhergehenden und gibt diese an ihre Neuronen, welche daraus wiederum ein Ergebnis berechnen. Dieses wird mit einer sogenannten Aktivierungsfunktion nochmal verändert und dann weitergegeben. Hierzu wird oft die ReLU-Funktion verwendet, welche nur positive Werte oder null für negative Werte weitergibt. Wozu eine Aktivierungsfunktion benötigt wird, ist im Abschnitt 2.2.7 ausgeführt.
- Output Layer

  Das Output Layer ist, wie der Name bereits verrät, für die Ausgabe zuständig. Die Form der Ausgabe ist von der jeweiligen Aufgabenstellung abhängig. Grundsätzlich gibt es in dieser Schicht ein Neuron pro Klasse. Neuronen geben ihre Ergebnisse als Logits weiter, welche auf den ersten Blick nicht gut vergleichbar sind für das menschliche Auge. Deswegen werden diese im Output Layer meist über eine Softmax-Funktion in Wahrscheinlichkeiten umgewandelt.

Ein Neuron ist eine Rechnung mit folgender Form: $x1 * w1 + x2 * w2 + .. + xn * wn + b$. x steht hierbei für ein Ergebnis aus der vorherigen Schicht, w für das Gewicht und b ist der Bias. Die Nummerierung zeigt, dass ein Neuron pro Verbindung zu einem seiner Vorgänger ein individuelles Gewicht verwendet.
Das Training des Modells bezeichnet den Vorgang, bei dem für die Gewichte und Bias-Werte der Neuronen die bestmöglichen Werte ermittelt werden, um mit dem Modell erfolgreich Daten klassifizieren zu können. Der Ablauf sieht folgendermaßen aus:

1. Forward Pass

   Eingaben werden durch das Neuronale Netz propagiert. Das bedeutet, sie durchlaufen einmal das gesamte Netz und das Modell gibt eine Vorhersage ab.

2. Loss Function

   Der Fehler, also die Abweichung von der Vorhersage zum richtigen Ergebnis, wird berechnet. Die Berechnung des Fehlers sieht je nach Aufgabe des Modells anders aus; bei binärer Klassifikation wird der Fehler folgendermaßen berechnet: $L = -(y * log(ypred) + (1 - y) * log(1 - ypred))$. y ist die tatsächliche Klasse und $ypred$ die vorhergesagte. Der Logarithmus

wird hier verwendet, um stark falsche Schätzungen besonders zu bestrafen. Würde der Fehler über reine Subtraktion berechnet werden, so würde das Modell sehr langsam lernen. Der Unterschied einer gravierend falschen Schätzung wie etwa 0.1 und einer sehr guten wie 0.95 bei Zielklasse 1, wäre für das Modell nicht auffällig genug um es schnell in die richtige Richtung zu lenken. Durch Verwendung des Logarithmus wird dieses Problem gelöst, denn eine Schätzung von 0.1 würde bei Berechnung der Differenz zur Zielklasse 0.9 ergeben und bei $-log(0.1)$ 2.303.

3. Backpropagation
   Der ermittelte Fehler wird rückwärts durch das Netzwerk propagiert. Dabei wird für jedes Gewicht und Bias bestimmt, wie stark diese zu dem Fehler beigetragen haben und sie werden dementsprechend angepasst. Ein Gewicht wird über folgende Formel angepasst: $wneu = walt - *L/w$. $wneu$ stellt das neu errechnet Gewicht dar und $walt$ das alte, ist der Faktor für die Gewünschte Größe der Änderung des Gewichts und durch L/w wird die notwendige Veränderung berechnet. Ist L/w positiv, so hat Gewicht zum negativen Ergebnis der Klassifizierung beigetragen und es wird daher verringert. Ist L/w negativ, so hat dieses Gewicht zur Verkleinerung des Fehlers beigetragen und wird daher erhöht.
4. Dieser Ablauf wird wiederholt, bis das Modell keine signifikanten Fortschritte mehr macht.

Die Leistung eines Modells kann über eine Confusion Matrix (siehe Abschnitt 2.2.6) veranschaulicht werden. Je nach Aufgabe des Modells ist ein anderer dieser Werte am wichtigsten. Bei der Branderkennung beispielsweise sind False Negatives besonders brisant, da dies bedeutet, dass ein tatsächliches Brandszenario nicht als solches erkannt wurde. Um die kritischen Fehlerfälle zu minimieren, können die Gewichte der Loss-Function angepasst werden, sodass gewisse Fehler stärker gewichtet werden als andere. Im Fall der Branderkennung wird dies so eingesetzt, dass das Modell eher zum Auslösen eines Alarms neigt, um das Risiko eines nicht erkannten Brandes zu minimieren.
Da die Eingabedaten für normale Neuronale Netze in keinem zeitlichen Kontext stehen, werden für Probleme bei denen die Reihenfolge der Daten eine Rolle spielt RNNs (siehe Abschnitt 2.2.7 verwendet. [15][16]

2.2.4 Klassifikation

Als Klassifikation wird die Zuordnung von Daten zu vordefinierten Klassen bezeichnet. Ein Modell lernt dies durch Supervised Learning (siehe Abschnitt 2.2.6). Diese Zuordnung der Daten

basiert auf Features (siehe Abschnitt 2.2.5). Im Fall dieser Arbeit spricht man von 'Binary Classification'. Diese Bezeichnung wird gewählt, wenn als Ergebnis eine von zwei Klassen möglich ist. [15]

2.2.5 Features

Als Features werden die Werte bezeichnet, aufgrund derer das Modell seine Klassifikation (siehe Abschnitt 2.2.4) berechnet. Ein Feature kann entweder numerischer oder kategorialer Natur sein. Oft müssen die Features in der Datenvorverarbeitung vor dem Modelltraining noch verändert werden, um sie besser vergleichbar zu machen. Beispielsweise könnte aus dem Geburtsdatum das Alter errechnet werden, um den Unterschied zwischen zwei Geburtsdaten ersichtlich zu machen. [15]

2.2.6 Supervised Learning

Beim Supervised Learning wird das Modell mit bereits gelabelten Trainingsdaten angelernt. Das bedeutet, die Daten wurden manuell durch den Ersteller des Datensatzes oder den Entwickler des Modells zu den Klassen zugeordnet. Die Daten werden nun dem Modell gemeinsam mit ihren Labeln überreicht und dieses soll den Zusammenhang zwischen den Daten und ihren Labeln erlernen. Nach Abschluss des Trainings wird die Parameter-Kombination mit dem geringsten Fehler-Wert für das Modell verwendet. Die Genauigkeit des resultierenden Modells wird meist über eine Confusion Matrix veranschaulicht, welche in ihrer elementaren Form folgende Werte enthält:

- True Positive -> Anzahl korrekter Klassifizierungen in der Zielklasse
- True Negative -> Anzahl korrekter Klassifizierungen in der Gegenklasse
- False Positive -> Anzahl falscher Klassifizierungen in der Zielklasse
- False Negative -> Anzahl falscher Klassifizierungen in der Gegenklasse

Der Idealfall sieht natürlich vor, dass ein Modell alle Daten richtig klassifiziert, jedoch ist dies oft unrealistisch und deutet meistens auf 'Overfitting' hin. Overfitting beschreibt einen Zustand, bei dem das Modell die Trainingsdaten zu gut gelernt hat und somit bei neuen Daten versagt. Um diesen Verdacht zu bestätigen muss jedoch zuerst die Leistung des Modells bei unbekannten Daten getestet werden. Erbringt das Modell bei neuen Daten ebenso gute Ergebnisse wie bei den Trainingsdaten so spricht man von guter Generalisierung.
[15]

2.2.7 LSTM

LSTMs sind eine Unterkategorie von RNNs, welche wiederum eine Form von Neuronalen Netzen (siehe Abschnitt 2.2.3) darstellen. RNNs werden zur Klassifizierung (siehe Abschnitt 2.2.4) von Daten verwendet, welche durch ihre Gegebenheit eine Verarbeitung in gewisser Reihenfolge fordern und in Bezug zu den Vorgängern und Nachfolgern stehen. Meist handelt es sich hierbei um Messwerte eines Sensors. Ein herkömmliches Neuronales Netz könnte zwar einzelne Messpunkte kategorisieren, jedoch gibt es keine Möglichkeit, dem Netz den Zusammenhang zwischen den einzelnen Messwerten anzulernen.
Hier kommen RNNs ins Spiel. Diese Art der Modelle analysiert Zeitreihendaten Wert für Wert. Mithilfe sogenannter 'Hidden States' ist es dem Modell möglich, einen Zusammenhang mit den vorhergehenden Messwerten herzustellen. Der 'Hidden State' wird folgendermaßen berechnet: $h_t = tanh(w_x * x_t + w_h * h_{t-1} + b)$. Die Variable t beschreibt den zeitlichen Kontext, also den Index in der Messreihe. w_x ist die Gewichtsmatrix, die mit dem aktuellen Eingabewert x_t multipliziert wird. w_h ist die Gewichtsmatrix, die mit dem bisherigen Hidden State h_{t-1} multipliziert wird. Der Bias b wird in Abschnitt 2.2.3 bereits erklärt. Die Gewichte werden in einer Matrix gespeichert, da ein Layer aus mehreren Neuronen besteht und jedes dieser Neuronen ein eigenes Gewicht besitzt. Die Aktivierungsfunktion wird benötigt, um komplexe Zusammenhänge darstellen zu können. Ohne eine solche Funktion würde sich das Ergebnis eines Neurons immer linear zur Eingabe verhalten und somit auch bei der Verbindung mehrerer Neuronen.
Bei der Backpropagation in einem RNN wird der Fehler-Wert über alle Zeitschritte rückwärts gerechnet. Da dieses 'Rückwärtsgehen' über Ableitungen erreicht wird, ist der Einfluss der früheren Hidden States auf den Fehler kaum bestimmbar. Bei wenigen Zeitschritten stellt dies noch kein großes Problem dar. Da in dieser Arbeit jedoch mit Zeitreihen von jeweils 600 Messwerten gearbeitet wird, wäre ein RNN ein äußerst ineffektiver Ansatz.
Die Lösung zu diesem Problem stellt das LSTM dar. Um die Auswirkungen früher Hidden States auf das Ergebnis über längere Zeitreihen zu erhalten werden in einem LSTM sogenannte 'Gates' und 'Cell-States' verwendet. Hiervon gibt es folgende:

1. Forget Gate
   Formel: $f_t = (w_f * [h_{t-1}, x_t] + b_f)$
   Nutzen: Das Forget Gate berechnet, wie viel des aktuellen Cell-States c_{t-1} behalten werden soll. Dadurch werden irrelevante Informationen nicht weiter berücksichtigt.

2. Input Gate
   Formel: $i_t = (w_i * [h_{t-1}, x_t] + b_i)$

Nutzen: Das Input Gate berechnet, wie stark neue Informationen x_t aufgenommen werden. Dadurch gelangen nur die relevanten Informationen in den neuen Cell-State.

3. Candidate Cell-State
   Formel: $cCan_t = tanh(w_c * [h_{t-1}, x_t] + b_c)$
   Nutzen: Der Candidate Cell-State dient zu Umformung der Eingabe in die Form eines Cell-State um im nächsten Schritt den tatsächlichen neuen Cell-State berechnen zu können.

4. neuer Cell-State
   Formel: $c_t = f_t \odot c_{t-1} + i_t \odot cCan_t$
   Nutzen: Hier werden nun die vorherigen Schritt kombiniert, um den neuen Cell-State zu berechnen.

5. Output Gate
   Formel: $o_t = (w_o * [h_{t-1}, x_t] + b_o)$
   Nutzen: Berechnet, was für den neuen Hidden State relevant ist.

6. neuer Hidden State
   Formel: $h_t = o_t \odot tanh(c_t)$
   Nutzen: Durch Verbindung des Output Gate mit dem neuen Cell-State wird der neue Hidden State berechnet.

$w * [h_{t-1}, x_t]$ ist eine verkürzte Schreibweise für $w_x * x_t + w_h * h_{t-1}$. Das Rechenzeichen $\odot$ bedeutet, dass zwei Vektoren elementweise miteinander multipliziert werden.
Durch die verschiedenen Gates, welche die Informationen in dem Cell-State kontrollieren, ist es dem LSTM nun möglich, Informationen über eine größere Anzahl an Zeitschritten hinweg zu transportieren. Dadurch lässt sich ein LSTM auf weitaus längere Zeitreihen als ein RNN trainieren, da der Fehler viel weiter zurückverfolgt werden kann. Somit werden bei der Klassifikation Ereignisse berücksichtigt, welche viel weiter in der Vergangenheit liegen als bei klassischen RNNs. [17]

2.3 Verwendete Entwicklungssysteme

2.3.1 Git

Git ist ein verteiltes Versionskontrollsystem (Distributed Version Control System) zur Verwaltung von Änderungen an Dateien, insbesondere Quellcode, das es ermöglicht, Änderungen am Projekt über die Zeit nachzuverfolgen und frühere Versionen wiederherzustellen. Somit können mehrere

Entwickler gleichzeitig am selben Projekt arbeiten, ohne sich gegenseitig zu überschreiben. Jeder Programmierer besitzt eine vollständige lokale Kopie des gesamten Repositorys, inklusive Versionshistorie, wodurch viele Arbeitsschritte lokal und ohne Verbindung zu einem Server durchgeführt werden können. Änderungen werden in sogenannten Commits gespeichert, um auch nach längerer Zeit und vielen Änderungen eine nachvollziehbare Versionskontrolle zu ermöglichen. Damit neue Funktionen und Änderungen unabhängig vom Hauptprojekt reibungslos getestet und entwickelt werden können, können Mitglieder des Repositorys sogenannte 'Branches' (Entwicklungszweige) erstellen und diese später, nach erfolgreichem Testen, wieder mit dem Hauptprojekt zusammenführen (Merge). Git ermöglicht dadurch das Experimentieren mit neuen Funktionen, ohne den stabilen Hauptstand des Projekts zu gefährden. Als zusätzliches Backup dient die verteilte Struktur, durch welche mehrere vollständige Kopien des Projekts existieren. Die Open-Source-Software ist auf hohe Geschwindigkeit, Effizienz und Zusammenarbeit in Teams ausgelegt und wird von vielen Softwareentwicklern weltweit verwendet. Bei MoboView wurde es ebenfalls eingesetzt, um durchgehend einen klaren Überblick über das große Ganze zu bewahren und neue Funktionen sicher testen zu können. [18]

Abbildung 16: Logo von Git [19]

2.3.2 Webstorm

WebStorm (Logo siehe Abbildung 17) ist eine integrierte Entwicklungsumgebung (IDE), die von JetBrains entwickelt wurde. Unterstützte Technologien sind unter anderem JavaScript, HTML5 (Siehe Abschnitt 2.4.7), Node.js (Siehe Abschnitt 2.4.2), Angular (Siehe Abschnitt 2.4.7), uvm. Der Haupt-Einsatzbereich befindet sich in der Entwicklung von Webapplikationen und Mobile-Apps. WebStorm basiert auf der IntelliJ IDEA. Die Hauptfunktionen sind die intelligente Codevervollständigung (Autocompletion), automatische Fehlererkennung und Vorschläge zur Verbesserung des Codes beziehungsweise Hinweise auf veraltete Code-Syntax. Außerdem bietet WebStorm eine integriertes Debugging sowie auch eine Versionskontrolle (Git) an. Auch die Zusammenarbeit in Echtzeit wird durch die WebStorm IDE erleichtert. Konfigurationen können auf Projektebene übertragen werden, aber auch individuelle Themes und Einstellungsmöglichkeiten werden angeboten. [20]

Abbildung 17: Logo von Webstorm
[21]

2.3.3 Visual Studio Code

Visual Studio Code (Logo siehe Abbildung 18) ist eine Eine Entwicklungsumgebung (IDE) von Microsoft, welche alle Schritte der Softwareentwicklung in einem Tool kapselt. Es können Anwendungen erstellt, getestet und gedebugged werden, ohne zwischen verschiedenen Tools wechseln zu müssen. Die Umgebung eignet sich für die Entwicklung von Desktop-, Web- und mobilen Anwendungen.

Visual Studio wird sowohl von Einzelentwicklern als auch von größeren Teams eingesetzt und eignet sich für Projekte unterschiedlicher Größenordnungen. Neben der kostenlosen Community-Version stehen auch kostenpflichtige Varianten wie Professional und Enterprise zur Verfügung, die zusätzliche Funktionen für Teamarbeit und komplexe Softwareprojekte bieten.

Abbildung 18: Logo von Visual Studio Code
[22]

2.3.4 Docker Desktop

Docker Desktop ist eine Softwareplattform für Entwickler zum Erstellen, Ausführen und Teilen containerisierter Anwendungen. Das Programm ist für Windows, macOS und Linux einfach installierbar und stellt eine komplette Docker-Entwicklungsumgebung bereit. Docker ermöglicht das Containerisieren von Anwendungen, sodass Programme inklusive aller Abhängigkeiten in einer isolierten Umgebung laufen, während ebenfalls verschiedene Programmiersprachen und Frameworks unterstützt werden. Docker Desktop enthält zentrale Docker-Komponenten wie die Docker Engine (Daemon), die Docker CLI, Docker Compose und Kubernetes. Um Container,

Images, Volumes und Builds verwalten zu können, wird eine grafische Benutzeroberfläche (Dashboard) zur Verfügung gestellt. Außerdem wird eine Kubernetes-Integration unterstützt, was es ermöglicht, Container-Cluster lokal zu orchestrieren und zu verwalten. All das ermöglicht das lokale Entwickeln, Testen und Ausführen von Container-Anwendungen, bevor sie in Produktionsumgebungen oder Cloud-Systemen deployt werden. Aufgrund der einfachen Installation und Nutzung von Docker Desktop wurde es bei MoboView verwendet, um die Datenbanken 'Microsoft SQL Server' und 'TimescaleDB' zu deployen, damit beispielsweise Temperaturdaten, Userdaten usw. einfach und problemlos persistiert werden können. [23]

Abbildung 19: Logo von Docker Desktop [24]

2.3.5 WSL

'Windows Subsystem for Linux', kurz 'WSL', ist das Feature des Windows-Betriebssystems zur Ausführung einer Linux-Umgebung direkt unter Windows. Es ermöglicht die Nutzung von Linux und Windows gleichzeitig auf demselben System, ohne dass eine separate virtuelle Maschine oder Dual-Boot-System benötigt wird. WSL erlaubt das Ausführen einer vollständigen GNU/Linux-Umgebung innerhalb von Windows und unterstützt dabei die meisten Linux-Kommandozeilenprogramme, Tools und Anwendungen. Somit können Linux-Programme unverändert (nativ) ausgeführt werden, damit Entwickler gleichzeitig Windows- und Linux-Werkzeuge verwenden können. WSL wurde speziell entwickelt, um eine nahtlose und produktive Entwicklungsumgebung bereitzustellen, weshalb es insgesamt die Entwicklung von plattformübergreifender Software und Web- oder Serveranwendungen erleichtert. Bei MoboView wurde WSL 2 verwendet, welches bei Docker Desktop [23] als Backend zum Einsatz kommt und beim Installieren der Anwendung automatisch mitinstalliert und eingerichtet wird. [25]

Abbildung 20: Logo von WSL [26]

2.4 Verwendete Technologien

2.4.1 Mobotix MX-M16TB-R079

Die im Projekt eingesetzte Wärmebildkamera vom Typ Mobotix MX-M16TB-R079 wurde dem Projektteam dankenswerterweise von der voestalpine zur Verfügung gestellt. Es handelt sich dabei um eine Dual-Sensor-Kamera mit zwei getrennten Linsen, die eine kombinierte Aufnahme im sichtbaren Spektrum sowie im Infrarotbereich ermöglicht. Dadurch können sowohl konventionelle Bildinformationen als auch thermische Daten gleichzeitig erfasst und ausgewertet werden. Die Kamera ist für industrielle Anwendungen ausgelegt und erlaubt die zuverlässige Messung von Temperaturen bis zu 550 °C, wodurch sie sich insbesondere für Hochtemperaturprozesse eignet. Weitere technische Details und Spezifikationen sind in der Herstellerdokumentation angeführt. [27]

2.4.2 Backend

Node.js

Node.js ist laut [28] eine plattformübergreifende, Open-Source JavaScript-Laufzeitumgebung. Da Node auf dem Single-Process-Modell beruht und außerhalb des Browsers mit einer V8-Engine läuft, wird die Effizienz gesteigert und eine hohe Performance erreicht. Standardmäßig werden I/O-Operationen asynchron ausgeführt und die Verarbeitung ohne Wartezeiten sofort fortgesetzt. Sobald das Ergebnis verfügbar ist, wird die zugehörige Callback-Methode bzw. das Promise verarbeitet, ohne den Main-Thread dabei zu blockieren. Somit sind mit wenig Overhead viele Verbindungen gleichzeitig möglich, während der Verzicht auf ein Thread-Management die Komplexität und die Fehleranfälligkeit reduziert. Da sich Node.js im Allgemeinen gut für Projekte dieser Art, insbesondere für die Verarbeitung und Verwaltung unterschiedlichster Ressourcen, eignet und Vorkenntnisse dazu bereits vorhanden waren, wurde es in diesem Projekt als Backend-Technologie in Kombination mit Typescript eingesetzt.

Abbildung 21: Logo von Node.js [29]

2.4.3 npm

npm steht für 'Node Package Manager' und ist der Standard-Paketmanager für Node.js, welcher verwendet wird, um JavaScript-Pakete bzw. Bibliotheken zu installieren, zu verwalten und zu veröffentlichen. Da npm auf eine zentrale Online-Datenbank (npm Registry) zugreift, in der Millionen von Paketen gespeichert sind, wird es Entwicklern ermöglicht, Code von anderen wiederzuverwenden und eigene Pakete zu veröffentlichen (publish), welche man dann über das Command Line Interface (CLI) installieren, aktualisieren und entfernen kann. npm verwaltet Projektabhängigkeiten (Dependencies) automatisch und sorgt dafür, dass alle benötigten Bibliotheken installiert werden. Diese Dependencies und Metadaten werden typischerweise in der package.json-Datei definiert. Es können jedoch nicht nur lokale Abhängigkeiten für ein Projekt verwaltet werden, sondern auch global installierte Werkzeuge. Außerdem ermöglicht die Versionsverwaltung von Paketen (Semantic Versioning) kompatible Updates, damit immer alles reibungslos funktionieren und angepasst werden kann. Projekt-Skripte, wie bei MoboView zum Beispiel 'npm run backend', um das Backend im Entwicklungsmodus mit Auto-Neustart zu starten, können ebenfalls über die Kommandozeile ausgeführt werden. npm ist Teil der Node.js-Installation und wird automatisch mitinstalliert. [30]

Abbildung 22: Logo von npm [31]

Express.js

Express.js ist ein Open-Source Web-Framework für Node.js zur Entwicklung von Webanwendungen und APIs und gehört zu den meistverwendeten Backend-Frameworks im Node.js-Ökosystem. Es wird zur Verarbeitung von HTTP-Anfragen/-Antworten verwendet und erleichtert die Entwicklung von Server-Side-Anwendungen. Dadurch wird eine schnelle Erstellung von Webanwendungen, REST-APIs und Backend-Services ermöglicht. Node.js selbst bietet nur die grundlegenden Funktionen zum Erstellen von Servern, deshalb sind viele typische Aufgaben der Webentwicklung ohne Framework aufwendig selbst zu implementieren. Express stellt deshalb genau dafür vorgefertigte Mechanismen und Strukturen bereit, wodurch das Definieren von Request-Handlern für verschiedene HTTP-Methoden und Routing, also das Zuordnen von URLs zu bestimmten Funktionen, ermöglicht werden. Außerdem können dynamische Antworten erzeugt werden, da Templates unterstützt und die Daten dann direkt in diese eingesetzt werden. Durch

das Middleware-System von Node.js können zusätzliche Funktionen implementiert werden, welche dann während der Verarbeitung einer Anfrage ausgeführt werden, um beispielsweise Cookies, Sicherheitsfunktionen, Logins, Sessions, URL-Parameter, POST-Daten uvm. zu verarbeiten. Zusätzlich zu alldem existiert ein großes Ökosystem an Erweiterungen und Bibliotheken, welche typische Webentwicklungsprobleme lösen, wie z.B. bei MoboView das Arbeiten mit JWTs, Datenbanken und Cron-Modulen. Aus diesem Grund und da bereits bekannt war wie man mit Express.js umgeht, wurde sich im Backend in Kombination mit Typescript dafür entschieden. [32]

Abbildung 23: Logo von Express.js [33]

REST-API

Eine REST-API ist eine Programmierschnittstelle, die den Prinzipien der REST-Architektur folgt. REST steht für 'Representational State Transfer' und wird verwendet, um verteilte Systeme und Anwendungen miteinander zu verbinden und somit den Zugriff auf Ressourcen wie Daten oder Funktionen eines Servers durch einen Client zu ermöglichen. Der Server und der Client sind voneinander unabhängig, jedoch benötigt der Client die URI der Ressource, um auf diese zuzugreifen. Außerdem haben alle Anfragen für dieselbe Ressource ein einheitliches Format. REST-APIs verwenden typische HTTP-Operationen wie GET zum Abrufen von Daten, POST zum Erstellen einer neuen Ressource, PUT, um eine bestehende zu aktualisieren, und DELETE, um sie zu löschen. Bei der Datenübertragung sendet der Server eine Repräsentation der Ressource an den Client, wobei die Daten in unterschiedlichen Formaten übertragen werden können. Im Fall von MoboView wird mit JSON gearbeitet. REST-Anfragen bestehen unter anderem aus Parametern und Headern, welche Informationen über Authentifizierung, Metadaten, Cookies und Caching enthalten, wohingegen die Antworten HTTP-Statuscodes zur Rückmeldung des Ergebnisses beinhalten. Für einen besseren Überblick über die gesamte API kann diese über Spezifikationen beschrieben und somit dokumentiert werden, um Informationen wie die verfügbaren Endpunkte, die möglichen Operationen, Parameter und Authentifizierungsmethoden festzuhalten. Da über Express.js bereits bekannt war, wie man mit REST-APIs arbeitet und dass sie eine saubere Architektur ermöglichen, wurden sie bei MoboView ebenfalls verwendet. [34]

2.4.4 OpenAPI

OpenAPI (OpenAPI Specification, OAS) ist ein Standard zur Beschreibung von HTTP- bzw. REST-APIs. Die Spezifikation definiert Struktur und Syntax einer API in einem standardisierten Format und durch die Programmiersprachenunabhängigkeit können APIs unabhängig von den verwendeten Technologien beschrieben werden. Die API-Beschreibung wird meist in JSON oder YAML geschrieben, wodurch eine OpenAPI-Datei die komplette API dokumentieren kann. Zur Dokumentation gehören u.a. verfügbare Endpunkte, Parameter für Anfragen, Request- und Response-Strukturen, Authentifizierungsmechanismen und zusätzliche Metadaten wie Kontaktinformationen oder Lizenzangaben. Durch die standardisierte Beschreibung können Menschen und Maschinen verstehen, wie eine API funktioniert, ohne den Servercode zu kennen, während Entwickler ebenfalls die Funktionen und Fähigkeiten eines Services schnell erkennen können. OpenAPI kann während des gesamten API-Lebenszyklus eingesetzt werden, um automatisch die Dokumentation für APIs und mit weiteren Tools auch Client-Bibliotheken und Server-Code generieren zu lassen. OpenAPI erleichtert dadurch Integration, Wartung und Zusammenarbeit zwischen dem Projektteam. [35]

Abbildung 24: Logo von OpenAPI [36]

2.4.5 JWTs

JSON Web Token ist ein offener Standard zur sicheren Übertragung von Informationen zwischen Parteien. Die Informationen werden als JSON-Objekt übertragen, während das Token kompakt, URL-sicher und selbstenthaltend bleibt. Im Backend von MoboView finden die Tokens ihre Anwendung bei der Authentifizierung und Autorisierung von Usern über die API, was allgemein ein sehr häufiger Anwendungsfall für diese ist. Ein JWT besteht aus drei Teilen, welche alle durch Punkte getrennt werden: dem Header, dem Payload und der Signature. Der Header enthält die Metadaten über das Token und definiert somit den Token-Typ und den Signaturalgorithmus, was in diesem Projekt JWT und HS256 entspricht. Der Payload enthält sogenannte Claims, welche Informationen über den Benutzer/Kontext speichern, wie z.B. die Benutzer-ID oder wer das Token ausgestellt hat bzw. empfangen soll. Header und Payload werden Base64-URL kodiert, um Sonderzeichen wie Klammern problemlos zu übertragen und diese nach der Übertragung wieder zu dekodieren. Die Signature wird mit einem kryptografischem Algorithmus erstellt und

dient zur Überprüfung der Integrität des Tokens, damit dieser nicht unbemerkt verändert werden kann. HS256 steht für HMAC mit SHA-256, welcher der symmetrische Signaturalgorithmus ist, der in diesem Projekt mithilfe eines gemeinsamen geheimen Schlüssels zum Signieren und Verifizieren von JWTs verwendet wird. Wenn sich ein User einloggt, bekommt dieser ein JWT, welches bei den meisten weiteren Requests mitgesendet wird, wobei der Server bei diesem die Tokenstruktur, Signatur mit Secret Key und die Claims überprüft, bevor der Client erfolgreich die Antwort vom Server erhält. [37]

RSA-basierte Entschlüsselung

RSA ist ein asymmetrisches Kryptoverfahren mit einem Schlüsselpaar, welches sich aus einem öffentlichen Schlüssel, der geteilt werden darf, und einem privaten Schlüssel, der geheim gehalten werden muss, zusammensetzt. Hierbei gilt, dass der Sender mit dem öffentlichen Schlüssel verschlüsselt und der Empfänger mit dem privaten entschlüsselt. Die Sicherheitsidee basiert darauf, dass ohne den privaten Schlüssel die Rückgewinnung des Klartexts praktisch nicht möglich ist. Der Standard RFC 8017 spezifiziert RSAES-OAEP, was auch bei MoboView im Backend für die Entschlüsselung zum Einsatz kommt. Außerdem wird bei der Implementierung auch Padding verwendet, um durch das Anhängen von Bytes eine verschlüsselte Nachricht auf eine gewisse Blockgröße zu strecken, um somit bekannte Angriffe noch besser verhindern zu können. Weitere Sicherheitsanforderungen für RSA-Implementierungen ist z.B. die Verwendung kryptografisch sicherer Zufallswerte, korrekte Parameterwahl und eine konsistente Fehlerbehandlung. Bei MoboView verschlüsselt der Client die Kamera-Zugangsdaten, bzw. Credentials, mit dem Public Key, während der Server diese dann wieder mit dem Private Key entschlüsselt. Da der private Schlüssel ausschließlich auf dem Server bleibt, können somit sensible Daten verschlüsselt übertragen und gespeichert werden. [38]

Microsoft SQL Server

Microsoft SQL Server ist ein relationales Datenbank-Managementsystem, das zur Speicherung, Verwaltung und Abfrage strukturierter Daten dient. Anwendungen und Tools können sich mit einer SQL-Server-Instanz verbinden, die anschließend Datenanforderungen verarbeitet, etwa das Speichern und Abfragen von Daten oder das Verarbeiten von Transaktionen. SQL Server besteht aus mehreren Kernkomponenten. Dazu zählt die Database Engine, welche diese Anforderungen umsetzt und parallele Zugriffe sowie Datenintegrität unterstützt. Weiters existiert eine Replikationsfunktion, um Daten zwischen mehreren Servern und Datenbanken zu kopieren oder abzugleichen. Zusätzlich stehen verschiedene Services zur Verfügung, um beispielsweise

Daten zu bereinigen, Analysen oder Berichte zu erstellen, aber auch welche, um die Daten mit maschinellem Lernen zu untersuchen. Darüber hinaus werden gleichzeitige Nutzerzugriffe unterstützt, die ACID-Kriterien für Transaktionen erfüllt und Tools wie SQL Server Management Studio zur Verfügung gestellt, um Administration und Entwicklung zu erleichtern. Mit der Datenbank wird per Transact-SQL, eine von Microsoft erweiterte SQL-Variante, kommuniziert. Das DBMS kann auf Windows, Linux, virtuellen Maschinen und in Containern installiert werden, wobei in diesem Projekt letzteres mithilfe von Docker Desktop umgesetzt wurde. Da bei MoboView Userdaten, Kameradaten und Sessions gespeichert werden müssen, eignet sich Microsoft SQL Server äußerst gut für diesen Anwendungsfall. [39]

Abbildung 25: Logo von Microsoft SQL Server [40]

PostgreSQL

PostgreSQL ist ein leistungsfähiges, objektrelationales Datenbanksystem, das SQL nutzt sowie erweitert und zudem Open Source ist. Dieses frei verfügbare ORDBMS kann auf allen wichtigen Betriebssystemen installiert und als Docker-Container verwendet werden. PostgreSQL ist besonders bekannt für die Zuverlässigkeit, Datenintegrität, Flexibilität, ACID-Konformität und robuste Architektur. Es existieren unzählige Features für Entwickler und Administratoren, um beispielsweise Datenmengen in verschiedensten Größen sicher zu verwalten und zu schützen. Dabei werden auch zahlreiche Features für Leistung, Parallelität, Recovery und Sicherheit zur Verfügung gestellt, wie Indexierung, Replikation, Authentifizierung und Rollenmodelle. Außerdem unterstützt das ORDBMS mehrere Programmiersprachen und bietet dabei eine hohe Erweiterbarkeit. Dadurch ist es möglich, neben den standardmäßigen Datentypen, Funktionen und Modulen auch eigene zu definieren. Da bei MoboView für die Branderkennung im Sekundentakt Temperaturdaten gespeichert werden, eignet sich PostgreSQL mit der Erweiterung TimescaleDB besonders gut dafür. Die Datenbank bietet eine sehr gute Unterstützung für

Zeitreihendaten, hohe Performance bei vielen Inserts, hohe Datenintegrität sowie gleichzeitiges Lesen und Schreiben ohne Lock-Probleme.[41]

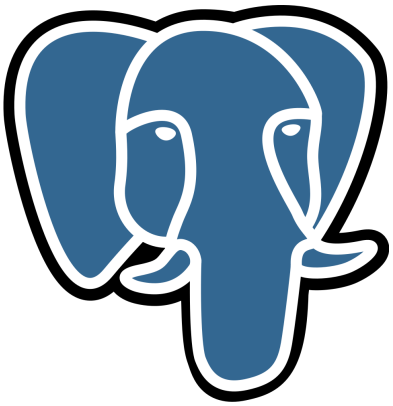


Abbildung 26: Logo von PostgreSQL [42]

2.4.6 TimescaleDB

Die Open-Source Time-Series-Datenbank TimescaleDB, wurde für die Verarbeitung zeitbasierter Daten entwickelt und als Erweiterung von PostgreSQL (siehe 2.4.5) implementiert. Sie unterstützt Standard-SQL und wurde speziell für Speicherung, Analyse und Verarbeitung großer Mengen zeitbasierter Daten entwickelt. Diese Zeitreihendaten bestehen aus Messwerten oder Ereignissen mit Zeitstempel. Im Fall von MoboView handelt es sich um Temperaturdaten, welche sekündlich fortlaufend gesendet und persistiert werden. Diese Daten werden in sogenannten Hypertables, eine spezielle Tabellenstruktur für Zeitreihendaten, chronologisch gespeichert. Da die Daten automatisch nach Zeitintervallen partitioniert und in kleinere Einheiten, sogenannte Chunks, aufgeteilt werden, müssen bei Abfragen nur relevante Zeitbereiche durchsucht werden, wodurch insgesamt eine höhere Performance erreicht wird. Außerdem gibt es auch Speicher- und Performance-Optimierungen, um die Speicherung und Verarbeitung der Daten mit gewissen Funktionen noch effizienter und schneller zu gestalten. Dazu tragen z.B. auch die Continuous Aggregates, wobei häufige Aggregationen wie Summen/Durchschnitte automatisch vorberechnet werden, und die Data Retention Policies, welche das automatische Löschen oder Verwalten alter Daten ermöglichen, stark bei. Da MoboView stark auf den großen Mengen an Zeitreihendaten aufbaut und diese auch effizient verarbeitet und persistiert werden müssen, wurde für diese Aufgabe TimescaleDB ausgewählt.

Abbildung 27: Logo von TimescaleDB [43]

2.4.7 frontend

Angular

Angular (Logo siehe Abbildung 28) ist ein leichtgewichtiges Frontend-Framework für Webapplikationen, welches von Google entwickelt wurde. Es basiert auf der Programmiersprache TypeScript (Siehe Abschnitt 2.4.7) und ermöglicht es, kurzerhand SPAs zu erstellen. Angular folgt einer komponentenbasierten Architektur und gewährleistet zugleich eine klare Trennung zwischen Darstellung (HTML) (Siehe Abschnitt 2.4.7), Logik (TypeScript) (Siehe Abschnitt 2.4.7) und Styling (SCSS) (Siehe Abschnitt 2.4.7). Es bringt viele eingebaute Funktionen mit sich wie beispielsweise Routing, Dependency Injection oder State-Handling. Das Frontend läuft gänzlich im Browser und kommuniziert mit dem Backend über REST-APIs. Es können UI-Updates ohne vollständiges Neuladen der Website erfolgen und ist aufgrund dessen besonders gut geeignet für interaktive Dashboards. Angular wurde bereits im schulischen Kontext verwendet und dadurch wurde schon praktische Erfahrung gesammelt, was ausschlaggebend für die Technologieentscheidung war. Auch die einfache Integrationsmöglichkeit von Charts (Siehe Abschnitt 2.1.1) hat maßgeblich zur Auswahl von Angular beigetragen.

Abbildung 28: Logo von Angular
[44]

HTML

HTML (Logo siehe Abbildung 29) ist das Fundament für die Umsetzung moderner Webapplikationen. Es definiert Struktur und Inhalt von Webseiten und beschreibt Elemente wie Überschriften, Bilder und Links. HTML basiert auf Markup-Tags, welche als Container für Inhalte fungieren. Die Syntax sieht folgendermaßen aus: Es gibt öffnende und schließende Tags, wobei inmitten deren Inhalt definiert wird. Diese einheitliche Struktur wird dann in Folge vom Browser interpre-

tiert, indem er den HTML-Code lesen und in visuell dargestellte Webseiten umwandelt. HTML bildet somit die Grundlage für weiterführende Webtechnologien. [45]

Abbildung 29: Logo von HTML [46]

SCSS

SCSS ist eine Erweiterung von CSS, welche zur effizienteren Gestaltung von Stylessheets dient. Der SCSS-Code wird vor der Nutzung in CSS kompiliert. Sprich der Browser rendert dann nur das CSS. Der Präprozessor SASS übernimmt die Aufgabe den geschriebenen SCSS-Code zu verarbeiten. Am Ende fällt ein standardkonformes CSS heraus (Siehe Abbildung 30). SCSS bringt allerdings auch wiederverwendbare Variablen mit sich um dieselben Farbwerte und Abrundungen applikationsweite zu gewährleisten (Siehe Abschnitt 2.4.7). Vorteile gegenüber reinem CSS sind die bessere Strukturierung, effizientere Entwicklung und Minimalisierung von redundantem Code. Außerdem ist die Nutzung in vielen Frameworks und Libraries möglich, was in umfangreicher Dokumentation mündet, und weshalb sich für SCSS entschieden wurde. [47]

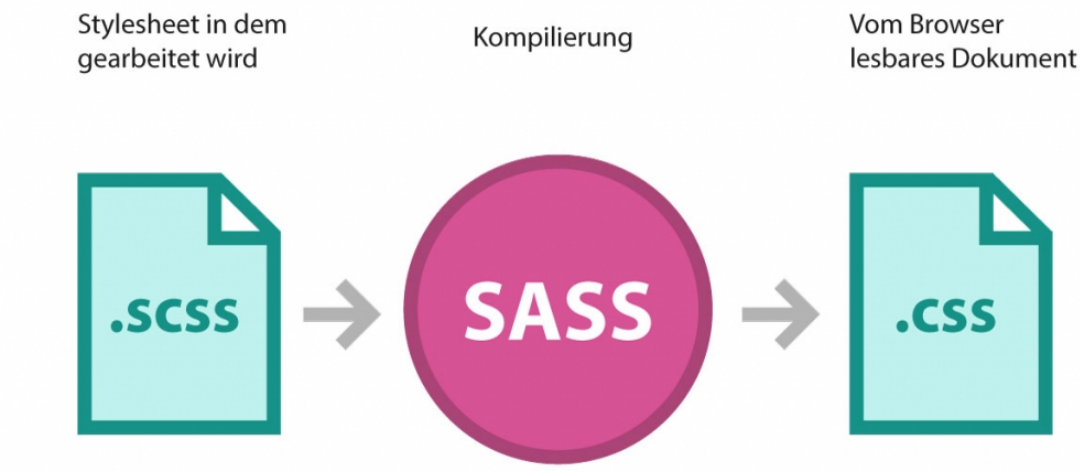


Abbildung 30: Kompilierung mittels SASS [47]

TypeScript

TypeScript ist eine von Microsoft entwickelte Skriptsprache, welche 2012 veröffentlicht wurde. Sie basiert auf JavaScript und erweitert es dahingehend, dass statische Typisierung mit eingebunden wird. TypeScript ist eine Obermenge von JavaScript was impliziert, dass jeder JavaScript-Code gleichzeitig auch gültiger TypeScript-Code ist (Siehe Abbildung 31). Außerdem ist TypeScript auch kompatibel mit JavaScript-Bibliotheken. TypeScript-Code wird vom Compiler zu einem vom Browser auszuführenden JavaScript-Code umgewandelt. Im Backend wurde ebenfalls hauptsächlich mit TypeScript gearbeitet. [48]

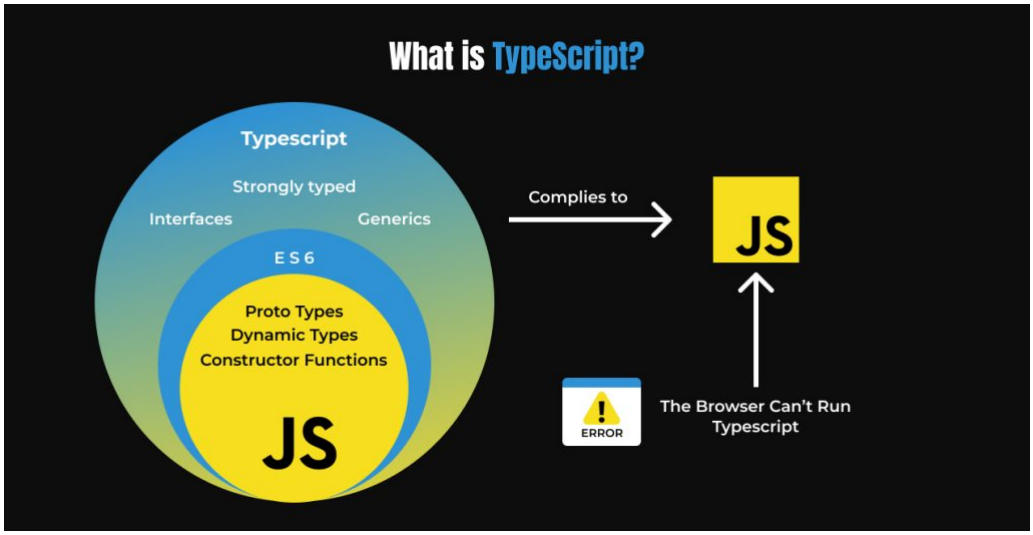


Abbildung 31: Typescript
[48]

2.4.8 branderkennungskomponente

Flask

Flask ist ein Webframework mit dem Ziel der schnellen und einfachen Entwicklung von Backends. Es wird auch oft als 'micro'-Webframework bezeichnet, was jedoch nicht auf eingeschränkte Funktionalität hindeutet, sondern die Philosophie hinter dem Aufbau von Flask beschreibt. Im Gegensatz zu einem Framework wie Django, welches sein gesamtes Angebot an Funktionen bereits bei der Installation mitbringt, kommt mit der Installation von Flask quasi nur das Nötigste. Das birgt natürlich Vor- und Nachteile. Einen Server mit Flask aufzusetzen ist einfacher, da auf keine Vorgaben hinsichtlich der Projektstruktur Rücksicht genommen werden muss. Jedoch müssen im Gegenzug Features wie Session-Handling oder ORM nachinstalliert werden. Flask arbeitet grundsätzlich synchron. Jeder eingehende Request bekommt einen 'Worker', also einen Thread (siehe Abschnitt 2.2.2) zugeordnet, welcher belegt ist, bis der Request vollständig bearbeitet wurde. Mittlerweile gibt es sogenannte 'async-views'; diese geben jedoch nicht die Rechenleistung des Worker über Nutzung einer Event-Loop frei, wie es bei echter Asynchronizität der Fall wäre, sondern sie ermöglichen nur das asynchrone Arbeiten innerhalb des Requests. Würden also durch die Bearbeitung des Requests mehrere Aufrufe einer anderen API notwendig, so könnten diese parallel geschehen; jedoch wird die Rechenleistung des Workers während der Wartezeit nicht für die Bearbeitung eines anderen Requests zur Verfügung gestellt. [49][50]

Python Threads

Für die Umsetzung des Flask Servers (siehe Abschnitt 4.1.6) wurde die Bibliothek 'threading' verwendet. Threads (siehe Abschnitt 2.2.2) in CPython werden durch den sogenannten 'Global Interpreter Lock' limitiert, welcher echte Parallelität verhindert. Als 'CPython' wird die am weitesten verbreitete Python Implementierung in C bezeichnet. Der GIL funktioniert wie ein Sprechball im Kindergarten. Nur jener Thread, der im Besitz des GIL ist, darf Berechnungen auf der CPU durchführen. In anderen Programmiersprachen wie Java können Threads ihre Berechnungen auf eigenen CPU-Kernen durchführen. Dies wurde jedoch in CPython nicht gemacht, um interne Prozesse wie die Referenzzählung für Objekte simpler zu halten.

Um Threads verwalten zu können, wird ein Event verwendet. Dieses ist nicht mit den Events aus der Webentwicklung zu verwechseln, bei denen auf gewisse Aktionen des Nutzers mit einem Event-Handler reagiert wird. Das Event der Thread-Bibliothek kann viel mehr mit einer Flag verglichen werden. Wie in Listing 10 gezeigt, wird ein Event zunächst einmal mit einer aussagekräftigen Variablenbezeichnung instanziiert. Die auszuführende Funktion eines Threads kann auf

zweierlei Arten festgelegt werden. Entweder über ein sogenanntes 'Callable' also eine Funktion wie ein Listing 10 Zeile 4 oder aber durch Überschreiben der `.run`-Methode des Threads, was über Vererbung möglich ist. Über den Parameter `args=` können nun die Parameter der in Listing 11 dargestellten Zielfunktion angegeben werden. In dieser kann nun über `stop_event.is_set()` überprüft werden, ob das Event gesetzt wurde und somit kann es als Abbruchbedingung genutzt werden. [51][52]

Listing 10: Anlegen eines Threads

```
1       def StartThread():
2           stop_event = Event()
3
4           thread = Thread(
5               target=methodName,
6               args=(stop_event, parameter2),
7               daemon=True,
8           )
9
10          threadsList[key] = {"thread": thread, "stop_event": stop_event, "name": name}
11
12      def StopThread():
13          entry = threadsList.pop(key)
14          entry["stop_event"].set()
```

Listing 11: Zielfunktion des Threads

```
1       def methodName(stop_event, parameter2):
2           while True:
3           if stop_event.is_set():
4               break
```

Python

Python ist eine sogenannte interpretierte Programmiesprache. Das bedeutet, Python-Code wird direkt durch den Interpreter ausgeführt, anstatt davor vollständig in Maschinencode übersetzt zu werden.

Python wird als höhere Programmiersprache verstanden, da großer Wert auf Lesbarkeit gelegt wird und ein hoher Abstraktionsgrad zu Maschinencode besteht. Es wird dynamische Typisierung unterstützt, was bedeutet, dass die Datentypen für Variablen nicht angegeben werden müssen. Da beim Erstellen eines Python Programms kein Projekt angelegt werden muss und durch die Abstinenz einer Projektstruktur einiges an Overhead wegfällt, wird Python oft für die Entwicklung von Prototypen oder zur Entwicklung von Scripts verwendet. Durch die Entwicklung von Bibliotheken wie Pandas (siehe Abschnitt 2.5.1) und Numpy (siehe Abschnitt 2.5.1) wurde Python immer populärer in der Datenverarbeitung und somit auch zum Training von ML-Modellen. Durch Flask (siehe Abschnitt 2.4.8) ist Python auch sehr attraktiv zur schnellen Entwicklung kleiner Webserver. Durch seine große Community ist Python gut geeignet für

Anfänger, da große Mengen an Anleitungen in Form von Text und Videos vorhanden sind. [53]

2.5 verwendete bibliotheken und plug-Ins

2.5.1 Branderkennungskomponente

Bokeh

Bokeh ist eine Python Bibliothek, die interaktive Visualisierung von Daten ermöglicht. In einem erstellten Graf kann mit dem Mausrad gezoomt und die Position mit einfachem Klick und Ziehen in die gewünschte Richtung verändert werden. Ein weiteres wichtiges Feature stellt die Möglichkeit zur Speicherung der erstellten Grafen als HTML-Datei dar. Dies wurde dazu verwendet, um die Darstellung direkt neben der zugehörigen CSV-Datei abzulegen. Dadurch konnte einiges an Arbeitszeit gespart werden, da die Grafik nicht jedes mal aufs neue erstellt werden musste, wenn sie benötigt wurde.[54]

Pandas

Pandas ist eine Python Bibliothek, die zur Datenverarbeitung tabellarischer Daten wie CSVs genutzt wird. Es werden zwei Formen der Datenspeicherung angeboten. Zum einen Series, welche durch ihre eindimensionale Struktur ähnlich zu herkömmlichen Arrays scheinen. Jedoch bieten sie den Vorteil, dass die enthaltenen Werte mit einem Index beschriftet werden können; beispielsweise können bei der Speicherung von Monatsumsätzen die einzelnen Werte direkt mit ihren dazugehörigen Monaten versehen werden. Die zweite Form stellt der Dataframe dar, der quasi wie ein zweidimensionales Array ist, jedoch mit vielen praktischen Hilfsmethoden versehen wurde, welche ihn von Arrays abhebt. Neben Funktionen zur Umformung der geladenen Daten ist auch das Einlesen der Daten, wie in Listing 12 gezeigt, deutlich leichter als bei einem Array. [55]

Listing 12: Einlesen einer CSV-Datei mittels Pandas

```
1
2  df = pd.read_csv(csv_path, names=['frame', 'timestamp', 'temperature'])
```

Numpy

Bei der Datenverarbeitung werden häufig Berechnungen getätigt welche nicht selten über einfache Strichrechnungen hinausgehen. Um nun nicht alle verwendeten Funktionen eigenhändig implementieren zu müssen wird Numpy verwendet. Neben mathematischen Funktionen werden auch Umformungsschritte wie Sortieren, Statistikoperationen wie der Mittelwert und eine Funktion `.random` für Zufallswerte angeboten. Letztere wurde bei der Generierung der Trainingsdaten eingesetzt (siehe Abschnitt 4.1.4). Weiters stellt die Bibliothek ähnlich wie Pandas ihre eigene Form der Datenhaltung zur Verfügung; die nd-Arrays, wobei `nd` kurz für n-dimensional ist. Ein weiteres wichtiges Merkmal ist die Vektorisierung. Diese beschreibt grundsätzlich die Form in der Operationen an Arrays im Code geschrieben werden. Typischerweise würden Arrays wie in Listing 13 gezeigt bearbeitet werden. Dank Vektorisierung kann dies nun aber auch so wie in Listing 14 aussehen. Durch Verwendung dieser erhöht sich nicht nur die Lesbarkeit des Codes sondern es kann bei vielen Operationen einiges an Leistung eingespart werden, da Numpy im Hintergrund häufig mit C-Code und Verfahren wie SIMD arbeitet. Durch ihr breit gefächertes Angebot an Funktionalität hat sich die Numpy-Bibliothek zu einem weit verbreiteten Standard etabliert. [56]

Listing 13: herkömmliche Bearbeitung eines Arrays

```
1  for i in range(len(a)):
2      a[i] = a[i] + 5
```

Listing 14: Bearbeitung eines Arrays mit Numpy Vektorisierung

```
1  a = a + 5
```

PyTorch

Pytorch ist eine Python Bibliothek zum Zweck von Machine Learning. Die zentrale Datenstruktur von Pytorch sind sogenannte 'Tensors'. Diese sind mehrdimensionale Arrays welche für Eingabe, die Speicherung von Gewichten und Ausgaben verwendet werden. Sie ähneln stark den Numpy-ndArrays (siehe Abschnitt 2.5.1), weshalb Pytorch oft in Kombination mit dieser Bibliothek benutzt wird. Es werden wichtige Funktionen zum Modelltraining, wie Backpropagation mittels eines einfachen Methodenaufrufs und Datenverarbeitungsmöglichkeiten durch den `Dataload`, welcher das Mischen und erstellen von Batches erheblich vereinfacht bereitgestellt. [57]

2.5.2 Backend

Axios

Axios ist ein Promise-basierter HTTP-Client für Browser und Node.js, der zum Senden von HTTP-Anfragen (z.B. GET, POST) und zum Empfangen von Serverantworten dient. Der Client nutzt im Browser 'XMLHttpRequest' und in Node.js das HTTP-Modul, wobei auch asynchrone Programmierung mit Promises (z.B. async/await) unterstützt wird. Außerdem gibt es sogenannte 'Interceptors', welche die Requests/Responses vor der Verarbeitung abfangen und verändern, um Operationen wie Authentifizierung oder Logging zu ermöglichen. Jedoch ist es auch möglich, dass Requests abgebrochen werden bzw. ein Timeout auftritt. Eine Transformation zur automatischen Anpassung von Request- und Response-Daten, eine Fehlerbehandlung, strukturierte Responses sowie eine automatische JSON-Verarbeitung für Serialisierung und Parsing sind ebenfalls integriert. Des Weiteren werden verschiedene Formate wie JSON, FormData und URL-encoded unterstützt, und Funktionen wie Query-Parameter-Handling, Progress-Tracking, Schutzmechanismen und die Möglichkeit zur Bandbreitenbegrenzung bereitgestellt. MoboView verwendet Axios im Backend, um HTTP-Anfragen an andere Dienste oder APIs zu senden und deren Antworten sauber zu verarbeiten. [58]

mssql, pg

'mssql' ist eine Node.js Client-Bibliothek für Microsoft SQL Server, die die Verbindung zu SQL Server Datenbanken über TCP/IP ermöglicht. Zu den zentralen Funktionen der Bibliothek gehören das Ausführen von SQL-Abfragen bzw. CRUD-Operationen, das Verwalten von Transaktionen und das Connection Pooling (Verbindungsmanagement). Es ermöglicht die Integration in Backend-Anwendungen, um mit den Datenbanken arbeiten zu können, und abstrahiert komplexe SQL-Kommunikation, was die Nutzung für Entwickler vereinfacht. 'pg' ist eine Sammlung von Modulen zur Interaktion mit PostgreSQL-Datenbanken in Node.js. Unterstützt werden verschiedene Programmierstile wie Callbacks, Promises und async/await. Zu den zentralen Funktionen der Bibliothek gehören die Verbindung zur Datenbank herstellen, SQL-Queries ausführen und Connection Pooling für parallele Anfragen. Dazu kommen auch noch erweiterte Features wie Prepared Statements, Streaming und Cursor für große Datenmengen, Typkonvertierung und LISTEN/NOTIFY für Echtzeit-Events. 'pg' ist bewusst low-level für eine hohe Kontrolle der Datenbank und keine starke Abstraktion. [59] [60] [61] [62] [63]

cors, dotenv, http-status-codes

'CORS' ist eine Middleware für Express/Node.js, die die HTTP-Header für Cross-Origin Resource Sharing setzt. Somit kann bestimmt werden, welche Domains (Origins) auf die API zugreifen dürfen, was wichtig für die Browser-Sicherheit bei Requests zwischen Domains ist. Außerdem kann konfiguriert werden, welche Origins, Methoden und Header erlaubt sind und auch, ob sie global oder nur für einzelne Routen aktiviert werden sollen. Allerdings bietet 'CORS' keinen direkten Schutz, da der Browser lediglich anhand der gesetzten Header entscheidet, ob ein Zugriff erlaubt ist oder nicht. Im Backend gibt es auch gewisse Konfigurationen, die gespeichert und zur Verfügung gestellt werden müssen. Dazu wurde bei MoboView 'dotenv' verwendet. Die Bibliothek lädt Umgebungsvariablen aus einer .env-Datei und speichert sie in process.env. Dabei wird die Konfiguration von Code getrennt, z.B. API-Keys, Ports, usw. Somit wird die Sicherheit verbessert und das Arbeiten in unterschiedlichen Umgebungen (dev, prod) erleichtert, ohne dass externe Abhängigkeiten entstehen. Um das Ergebnis einer Anfrage zwischen Client und Server zu zeigen, wurden mit der Bibliothek 'http-status-codes' standardisierte HTTP-Antwortcodes verwendet. Beispiele für solche Codes wären 200 für 'OK', 404 für 'Not Found' oder 500 für 'Internal Server Error'. Diese Status-Rückmeldungen sind für die Kommunikation und das Fehlerhandling in APIs besonders wichtig. [64] [65] [66]

bcrypt, crypto

'bcrypt' ist eine Bibliothek mit Passwort-Hashing-Funktionen, um Passwörter nicht im Klartext übertragen oder speichern zu müssen. Hashing ist eine unumkehrbare Einwegfunktion, die automatisch den gehashten Wert mit einem Salt kombiniert, um z.B. selbst bei gleichen Passwörtern unterschiedliche Hashes zu erzeugen und sich vor Rainbow Tables schützen zu können. Außerdem ist der Algorithmus bewusst langsam, damit Brute-Force-Angriffe ineffizient werden. 'bcrypt' besitzt einen Cost-Factor, welcher die Anzahl der Hash-Runden bestimmt und mit steigender Hardware-Leistung erhöht werden kann, um den Algorithmus an zukünftige Rechenleistung anzupassen. Außerdem basiert die Bibliothek auf dem sogenannten 'Blowfish'-Algorithmus, wobei das Passwort zusammen mit dem Salt gehasht und gespeichert wird und beispielsweise beim Login dann der Hash des eingegebenen Passworts mit dem gespeicherten Hash verglichen wird. Das Ziel von 'bcrypt' sind sichere Authentifizierungssysteme. 'crypto' ist ein integriertes Node.js Modul, das kryptografische Funktionen wie Hashing, HMAC, Ver-/Entschlüsselung, digitale Signaturen und Verifikation bereitstellt. Intern basiert es auf OpenSSL und unterstützt symmetrische sowie asymmetrische Verschlüsselung. Außerdem ermöglicht es mittels 'random-Bytes' das Generieren sicherer Zufallszahlen. Typische Anwendungsfälle sind Passwort-Hashing,

Token/Session-Generierung, Datenverschlüsselung, uvm. Die Hashes werden deterministisch generiert, sie haben eine feste Länge, sind nicht umkehrbar und bei kleinen Änderungen kommt es bereits zu großen Unterschieden im Output. [67] [68]

node-cron

'node-cron' ist eine Bibliothek für zeitgesteuerte Aufgaben in Node.js. Sie basiert auf dem Unix-Cron-Scheduler, ein bekanntes System zur Automatisierung von gewissen Aktivitäten einer Applikation, und ermöglicht die automatische Ausführung von Funktionen zu einer bestimmten Zeit oder in Intervallen. Eingesetzt wird 'node-cron' direkt innerhalb der Anwendung, was eine einfachere Integration in die Backend-Logik und höhere Flexibilität ermöglicht. Der Hauptzweck der Bibliothek ist die Automatisierung von wiederkehrenden Aufgaben, wie z.B. Backups, das Versenden von E-Mails, Daten aktualisieren oder diverse Wartungsaufgaben. Da keine manuelle Ausführung mehr nötig ist, wird automatisch die Effizienz und Zuverlässigkeit einer Anwendung bzw. eines Workflows gesteigert. Zur Zeitplanung wird die Cron-Syntax verwendet, um die Ausführung einer Aktivität auf die Minute, Stunde, Tag, Monat, usw. genau einstellen zu können. Besonders für Backend-Prozesse, die beispielsweise wöchentlich durchgeführt werden müssen, oder zur Automatisierung von Web-Anwendungen eignet sich 'node-cron' äußerst gut. [69]

swagger-jsdoc, swagger-ui-express

'swagger-jsdoc' generiert automatisch eine OpenAPI-Spezifikation aus Code-Kommentaren und ermöglicht dadurch Dokumentation direkt im Quell-Code, ohne separate YAML/JSON-Dateien zu benötigen. Die Bibliothek nutzt definierte Optionen, wie z.B. einen Pfad zu allen APIs eines Projekts, um relevante Dateien zu scannen und daraufhin ein Swagger/OpenAPI-Objekt zu erzeugen, das die gesamte API beschreibt, mit Infos wie API-Titel, Version, Beschreibung, Server-URLs, Endpunkte und Methoden. Das Ziel von 'swagger-jsdoc' ist es, laufend eine automatisierte und konsistente API-Dokumentation zur Verfügung stellen zu können. Um die API-Dokumentation mithilfe einer grafischen Web-Oberfläche nutzen zu können, wurde 'swwagger-ui-express' entwickelt. Die Bibliothek basiert auf der OpenAPI-Spezifikation, um strukturierte API-Daten anzeigen zu können, und wird in Express als Route, durch z.B. '/api-docs', eingebunden. Die Weboberfläche ermöglicht somit eine Visualisierung aller Endpunkte, Anzeige von Requests/Responses und direktes Testen der API im Browser, während die Verständlichkeit der API und die Zusammenarbeit zwischen Backend und Frontend verbessert wird. Außerdem ist die 'Swagger UI' vollständig im Browser nutzbar, ohne die Implementierungslogik kennen zu müssen. [70] [71]

2.5.3 Frontend

RxJS

RxJS (Logo siehe Abbildung 32) liefert die Grundlage für einen reaktiven Programmier-Stil. Asynchrone Vorgänge wie HTTP-Requests, Benutzerinteraktionen oder UI-Events werden als Observables modelliert. Durch Operatoren wie `.map`, `.filter`, `.switchMap` oder `.catchError` lassen sich Datenströme transformieren und Fehlerbehandlung abbilden. Zusätzlich stellt RxJS mit `Subject` und `BehaviorSubject` Mechanismen bereit, um Ereignisse aktiv auszulösen oder den aktuellen Zustand einer Datenquelle zu halten.

Abbildung 32: Logo von RxJS
[72]

CryptoJS

CryptoJS (Logo siehe Abbildung 33) stellt kryptografische Funktionen wie beispielsweise Hashfunktionen, symmetrische Verfahren und Hilfsfunktionen zur Codierung bereit. CryptoJS wird häufig für Hashing-Operationen verwendet, also überall dort, wo ein Klartext später nicht wieder benötigt wird.

Abbildung 33: CryptoJS Logo
[73]

Node forge

Node Forge (Logo siehe Abbildung 34) ist eine Kryptografie-Bibliothek, welche eine breite Palette an Verfahren unterstützt. Sie inkludiert Hashing und symmetrische sowie auch asymmetrische Verschlüsselung. Außerdem werden RSA- und Padding/Encoding- Verfahren unterstützt, wodurch sich Daten mit einem Public Key verschlüsseln lassen.

Abbildung 34: Node Forge Logo
[74]

Chart.js

Chart.js (Logo siehe Abbildung 35) ist eine Open-Source-JavaScript-Bibliothek, welche es ermöglicht, interaktive Charts in Webapplikationen einzubauen. Diagramme werden direkt im Browser gerendert und somit ist keine zusätzliche Serverleistung notwendig. Chart.js unterstützt eine Vielzahl an Diagrammtypen wie beispielsweise Linien-, Balken- und Kreisdiagramme sowie auch Scatter-Plots und Bubble-Charts. Außerdem bietet diese Bibliothek einige interaktive Funktionen, unter anderem Tooltips, Hover-Effekte und Animationen. Charts können benutzerspezifisch angepasst werden und zwar über Konfigurationsparameter. Dies wären beispielsweise Farben, Labels und Achsen. Der Nutzen der Bibliothek sollte darin liegen komplexe Daten so visualisieren zu können, dass Informationen für den Nutzer schnell ersichtlich werden. [75]

Abbildung 35: Logo von Chart.js
[76]

Ngx-scrollbar

Ngx-scrollbar (Logo siehe Abbildung 36) ist eine Bibliothek, welche die Darstellung einer anpassbaren Scrollbar ermöglicht. Sie wird verwendet ,um scrollbare Inhaltsbereiche im Frontend zu gewährleisten. Zum Einsatz kommt diese auf der Dashboard und Manage-User Seite um, wenn ein Überlauf entstehen sollte, den Inhalt im Bildschirm-Höhenbereich scrollbar zu machen.

Die Bibliothek wurde mittels

Listing 15: Installation von Ngx-Scrollbar

```
npm install ngx-scrollbar
```

installiert.

Im HTML wurde dann folglich der darzustellende Inhalt mit nachstehendem Element umschlossen (Siehe Listing 16),

Listing 16: Einbingung der Scrollbar

```
<ng-scrollbar>
  <!-- Inhalt -->
</ng-scrollbar>
```

um das Scrollen gewährleisten zu können.

Abbildung 36: Logo von NGX-Scrollbar

2.6 verworfene Optionen

2.6.1 Trainingsdaten durch Data Augmentation

Aufgrund der beschränkten Zeit die dem Projektteam in der Werkstatt (siehe Abschnitt 4.1.1) zur Verfügung stand und den ebenfalls beschränkten Mitteln zur Simulation der Brandszenarien war es nicht möglich genügend reale Daten zu sammeln. Konkret mangelte es an Anzahl und Variation der Daten. Es wurden zwar gute Szenarien definiert, jedoch stand die Ausrüstung und die, zur Bedienung dieser notwendigen, Mitarbeiter nicht ausreichend lange zur Verfügung um die Anforderungen für ein vollständiges, qualitatives Trainings-Dataset zu erfüllen. Dieses Problem ließe sich durch Data Augmentation (siehe Abschnitt 2.2.1) lösen, aber selbst hierfür konnten nicht ausreichend Daten gesammelt werden.

3 Ergebnis

In diesem Kapitel wird das Ergebnis im Überblick behandelt.

3.1 Architektur

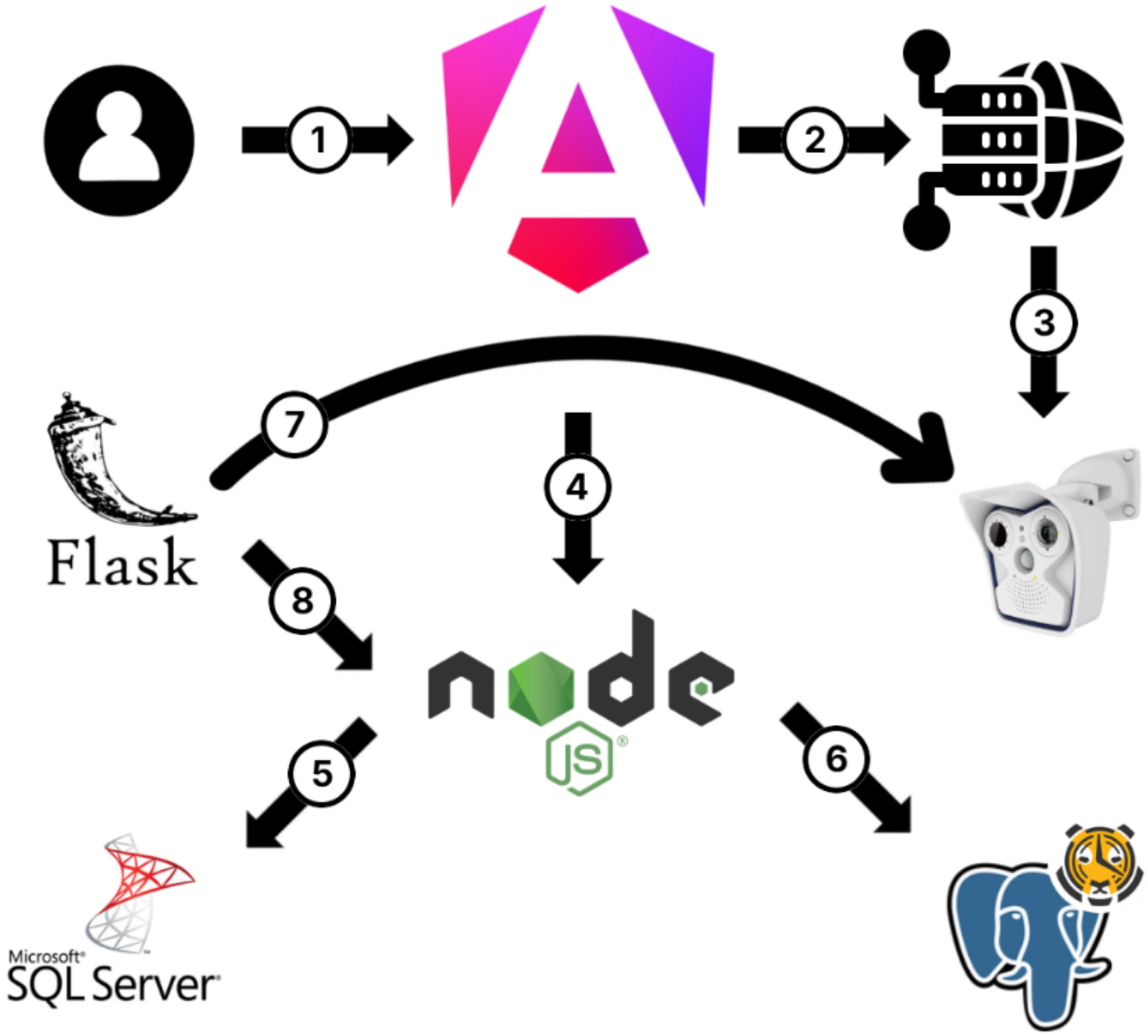


Abbildung 37: Architektur des MoboView Systems

(1)

Benutzer interagieren über die Angular-Webapplikation mit dem MoboView-System.

(2)

Alle Kamera-Feed-spezifischen Requests werden über einen Proxy geschleift.

(3)

Der Proxy handhabt dementsprechend die Einholung der Kamera-Feeds.

(4)

Das Frontend kommuniziert über Endpunkte mit dem Backend.

(5)

Das Backend legt User-Daten, deren Sessions, Kamera-Daten, Verlinkungen und Rollen-Requests in einer MSSQL Datenbank ab.

(6)

In PostgreSQL mit Timescale-Aufsatzt werden Zeitreihendaten, genauer gesagt historische Temperaturdaten, abgesichert.

(7)

Der Flask-Server auf dem das LSTM ausgeführt wird, holt alle Kameras des Systems ein, liest je Kamera kontinuierlich die Temperaturdaten aus und sendet diese klassifiziert an das Backend.

(8)

Die klassifizierten Temperaturdaten werden dann dementsprechend in der PostgreSQL Datenbank persistiert.

3.2 Frontend

Das Frontend des MoboView Systems dient als Benutzerschnittstelle. Benutzer können sich einloggen bzw. sich registrieren, um auf ihre Funktionen zuzugreifen. In der globalen Kameraliste sehen sie dann alle Kameras, die bis dato registriert wurden und können diese in das benutzerspezifische Dashboard übernehmen, um live sowie historische Temperaturdaten analysieren zu können. Schlägt das LSTM-Modell Alarm, wird der aktuelle Temperaturwert dementsprechend hervorgehoben und die historische Temperaturkurve im Zeitraum der Alarmdauer gelb eingefärbt,

um vergangene Alarme einsehbar zu machen. Zusätzlich bietet die Oberfläche direkte Kameraaktionen wie das Deaktivieren eines Alarms und das Wechseln der Linse auf Kamera-Ebene.

3.3 Backend

Das Backend von MoboView ist ein in TypeScript entwickelter Express-Server mit klarer Schichtenarchitektur aus Router-, Middleware-, Service- und Repository-Ebene. Es verwaltet Benutzer, Rollen, Kameras und temperaturbasierte Risikomessungen über eine Dual-Database-Strategie mit Microsoft SQL Server für Stammdaten und TimescaleDB für Zeitreihen. Sicherheit wird durch JWT mit serverseitigem Session-Store, rollenbasierte Zugriffskontrolle sowie API-Key-geschützte interne Endpunkte umgesetzt. Zusätzlich werden sensible Kameradaten verschlüsselt gespeichert. Beim Start sowie periodisch über Cron-Jobs führt das System automatische Bereinigungs- und Konsistenzprüfungen durch, um Datenqualität und Betriebssicherheit zu erhöhen. Ergänzt wird die Komponente durch OpenAPI/Swagger-Dokumentation, wodurch Implementierung und Schnittstellen nachvollziehbar dokumentiert sind.

3.4 Branderkennungskomponente

Im Flask-Server ist die Branderkennungs-Logik des Backends ausgelagert. Er liest kontinuierlich Temperaturframes der Wärmebildkamera aus, verarbeitet diese zu Sequenzen und klassifiziert sie mithilfe eines LSTM-Modells (siehe Abschnitt 2.2.7) als sicher oder riskant. Die Ergebnisse werden zur Persistierung an das eigentliche Backend weitergegeben.

4 Implementierung

In diesem Kapitel wird die Herangehensweise zur Implementierung Praxisteils behandelt.

4.1 Branderkennungskomponente

4.1.1 Testumgebung

Die Trainingsdaten stellen einen zentralen Faktor beim Training des Modells dar. Um eine möglichst realitätsnahe Abbildung echter Brandereignisse zu erreichen, wurde zunächst evaluiert, welche Möglichkeiten zur Erzeugung geeigneter Datensätze bestehen.

Auf Basis dieser Analyse wurde eine Unterscheidung in zwei Szenarien getroffen: ein rasch auftretender Brand, sowie ein sich langsam entwickelnder Brand.

Zur Simulation dieser Szenarien wurden eine Autogenschweißanlage, ein Halogenscheinwerfer sowie fachliche Unterstützung durch einen Mitarbeiter der Brandmeldetechnik eingesetzt. Die Kamera wurde an einer Betonsäule montiert und auf die definierte Teststation ausgerichtet.

Zur Simulation eines rasch auftretenden Brandereignisses wurde mithilfe einer Autogenschweißanlage ein Metallblech erhitzt (siehe Abbildung 38). Um einen langsamen Brand zu imitieren wurde ein Halogenscheinwerfer verwendet, da dieser durch seine hohe Leistung stark erhitzt.

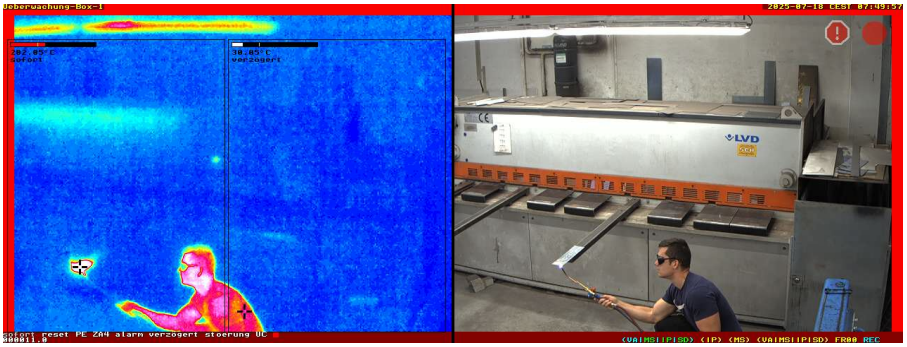


Abbildung 38: Blech wird erhitzt

Die entstehenden Temperaturverläufe wurden dabei kontinuierlich durch den Infrarotsensor der Kamera erfasst und mittels eines Python Scripts festgehalten (siehe Abschnitt 4.1.2) und visualisiert (siehe Abschnitt 4.1.3).

4.1.2 Datenerfassung

Die Gewinnung der Temperaturdaten stellte eine große Herausforderung dar, da die, von der Kamera gehostete, API keine ausführliche Dokumentation aufwies. Es wurde kein eigener Endpunkt zur Gewinnung der Live-Daten gefunden, jedoch gibt es in den Einstellungen auf der Web-Oberfläche der Kamera (siehe Abbildungen 39,40) eine Option, um die Daten des Thermalsensors mit den Bilddaten in deren Kommentaren mitzusenden.

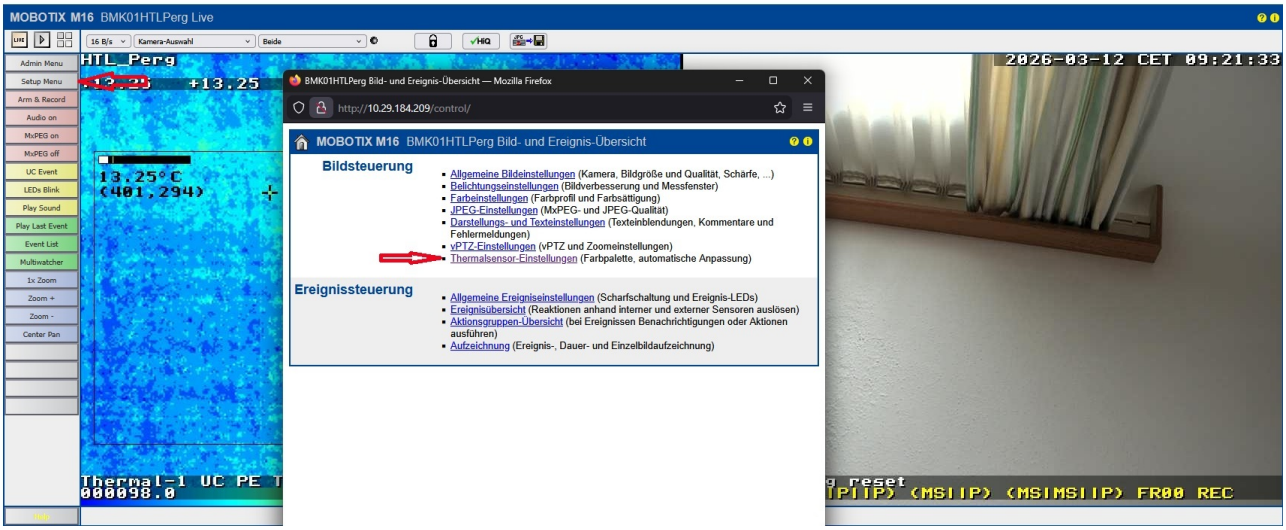


Abbildung 39: Setup Menü -> Thermalsensor Einstellungen

Abbildung 40: Thermal Rohdaten aktiviert

Die URL zur Datengewinnung ist in Listing 17 dargestellt. Wichtig sind folgende zwei Parameter:

- `snapimage`: ermöglicht Momentaufnahme des Streams, wodurch Daten nicht bei jedem Frame sondern nach Ermessen des Entwicklers abgefragt und gespeichert werden können. Im Fall von Moboview passiert dies alle 10ms.
- `showtsr`: TSR steht für thermal sensor reading und mit =1 aktiviert man das Mitsenden dieser Daten Clientseitig. [77]

Listing 17: URL des faststream-Endpunkts

```
1 STREAM_URL = 'http://10.23.232.147/control/faststream.jpg?snapimage&showtsr=1'
```

Ein Kommentarsegment kann an dem Marker `0xFF 0xFE` erkannt werden, wie in Zeile 5 Listing 18 illustriert. Die zwei nächsten Bytes geben Auskunft über die Länge des folgenden Kommentars. Hierbei ist zu beachten, dass der Kommentar-Marker nicht zur Länge zählt die Längeninformation selbst jedoch schon. [78]

Die Funktion versucht nun den Inhalt des Kommentars als UTF-8 Text zu parsen und überspringt ungültige Zeichen aufgrund des Parameters `errors='ignore'`.

Listing 18: Auslesen des JPEG Kommentars

```
1  def extract_jpeg_comment(jpeg_bytes):
2      comments = []
3      i = 0
4      while i < len(jpeg_bytes) - 1:
5          if jpeg_bytes[i] == 0xFF and jpeg_bytes[i+1] == 0xFE:
6              length = (jpeg_bytes[i+2] << 8) + jpeg_bytes[i+3]
7              segment = jpeg_bytes[i+4:i+2+length]
8
9              decoded = segment.decode('utf-8', errors='ignore')
10             comments.append(decoded)
11             i += 2 + length
12         else:
13             i += 1
14     return '\n'.join(comments)
```

Die erhaltenen Temperaturdaten werden in ein CSV file geschrieben und daraufhin visualisiert (siehe Abschnitt 4.1.3).

4.1.3 Visualisierung

Die Temperaturdaten werden, wie in Listing 19 gezeigt, in einen Pandas Dataframe (siehe Abschnitt 2.5.1) geladen und mittels bokeh (siehe Abschnitt 2.5.1) visualisiert (siehe Abbildung 41). Mit der Funktion `output_file` kann die erstellte Grafik als HTML-Datei abgespeichert werden. Die allgemeinen Einstellungen zur Grafik wie Achsenbeschriftung und Größe werden als

Parameter bei der Erstellung übergeben. Über das Attribut `.line` können nun einzelne Spalten des Dataframs auf die x- und y-Achse des plots aufgetragen werden.

Listing 19: Visualisierung mittels Bokeh

```
1
2  df = pd.read_csv(csv_path, skiprows=2, names=['frame', 'timestamp', 'temperature'])
3  df['temperature'] = pd.to_numeric(df['temperature'], errors='coerce')
4
5  output_file(f'{sequence_name}_temperatur.html')
6  p = figure(title=f'Temperaturverlauf - {sequence_name}',
7             x_axis_label='Frame', y_axis_label='Temperatur [C]',
8             width=1200, height=400)
9  p.line(x=df['frame'], y=df['temperature'], line_width=2, color='red', legend_label='TSR
       L1')
10 p.legend.location = 'top_left'
11 p.grid.visible = True
12 show(p)
```

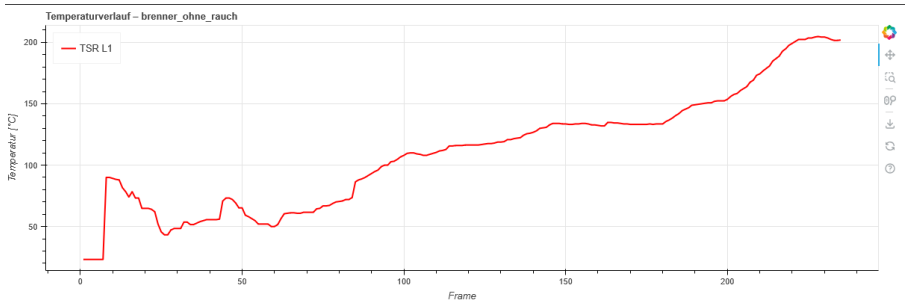


Abbildung 41: Temperaturverlauf des erhitzten Blechs

4.1.4 Trainingsdaten

Der erste Ansatz war es, die Trainingsdaten mittels Data Augmentation (siehe Abschnitt 2.2.1) zu erzeugen. Dies wurde jedoch verworfen, da zu wenig echte Daten vorhanden waren (siehe Abschnitt 2.6.1). Sind die Beobachtung aus der realen Welt selbst für Data Augmentation zu gering an der Zahl, so müssen die Trainingsdaten künstlich erzeugt werden. Basierend auf den Erkenntnissen aus den Aufzeichnungen (siehe Abschnitt 4.1.2) in der Testumgebung (siehe Abschnitt 4.1.1) wurden verschiedene Arten von riskanten und sicheren Temperaturverläufen zur Datengenerierung festgelegt.

In Listing 20 sind die grundsätzlichen Parameter zur Generierung zu sehen.
Aus `NUM_FRAMES / FRAME_INTERVAL_MS` ergibt sich die Länge der erstellten Sequenz; in diesem Fall zehn Minuten. Die Maximaltemperatur ergibt sich aus der Spezifikation der Kamera [27] in der diese mit 550°C beschrieben wird. Mit `BASE_RANGE` wird ein Wertebereich für den Start einer Sequenz bestimmt. Als Referenz für die Obergrenze von `SAFE_BASE_RANGE` wird die zulässige Betriebstemperatur für Wälzlager von 120°C laut [79] verwendet. Die Obergrenze für `RISKY` wurde bewusst höher gewählt, um den gesamten, als riskant eingestuften, Temperaturbereich abdecken zu können. Weiters wurde noch Noise verwendet, um natürliche Temperaturschwan-

kungen bei Messungen nachzuahmen. Sämtliche Berechnungen der nachfolgenden Code-Snippets wurden mit Numpy (siehe Abschnitt 2.5.1) umgesetzt.

Listing 20: Parameter zur Datengenerierung

```
NUM_FRAMES = 6000
FRAME_INTERVAL_MS = 100

TEMP_MIN = -20
TEMP_MAX = 550
SAFE_BASE_RANGE = (0.0, 120.0)
RISK_BASE_RANGE = (0.0, 450.0)

SAFE_NOISE_STD = 0.15

RISK_NOISE_STD_LOW = 0.3
RISK_NOISE_STD_HIGH = 2.5
```

sicherer Verlauf

Die wichtigste Variante eines sicheren Verlaufs ist in Abbildung 42 zu sehen. Hier wurde die Möglichkeit abgebildet, dass ein Wärmerer Gegenstand bzw. Organismus für kurze Zeit im Bild ist.

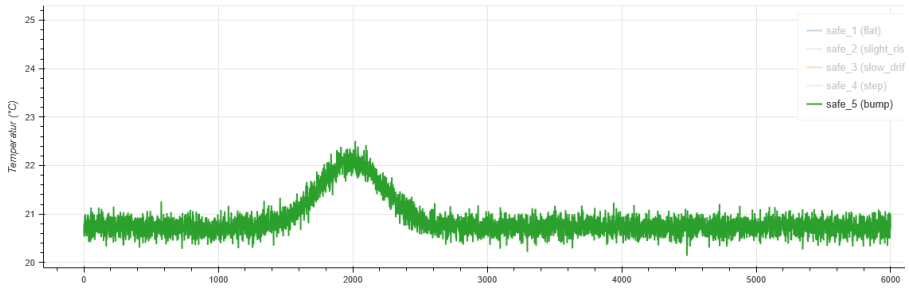


Abbildung 42: sichere Sequenz mit kleiner Erhebung

Erzeugt wurde dieser Verlauf mit dem in Listing 21 gezeigten Code. Über `SAFE_BUMP_AMP_RANGE` wird die Höhe der Erhebung bestimmt. Mit `center` wird eine zufällige Positon in der Sequenz berechnet, an der sich die Erhebung befinden soll. `width` gibt die Breite der Erhebung an; bei einem kleinen `width`-Wert gestaltet sich die diese spitz und bei einem großen Wert eher flach.

Listing 21: Generierung einer Safe Sequenz mit kleiner Erhebung

```
    # SAFE_BUMP_AMP_RANGE = (0.3, 1.5)
    elif selected_pattern == 'bump':
            center = np.random.randint(NUM_FRAMES // 4, 3 * NUM_FRAMES // 4)
            width = np.random.randint(50, 300)
            amp = np.random.uniform(*SAFE_BUMP_AMP_RANGE)
            x = np.arange(NUM_FRAMES)
            bump = amp * np.exp(-0.5 * ((x - center) / width) ** 2)
            temps = temps + bump + noise
```

riskanter Verlauf

Mit dem in Abbildung 43 gezeigten Verlauf, wird die Überhitzung einer Anlage modelliert.

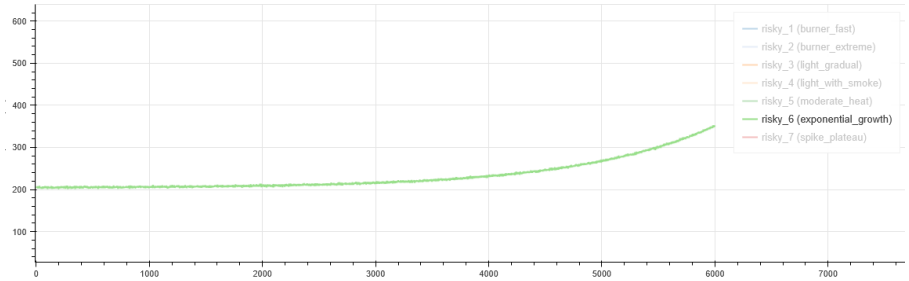


Abbildung 43: riskante Sequenz mit exponentiellem Anstieg

Diese Temperatursequenz wird durch den Code aus Listing 22 generiert. In `total_rise` wird ein zufälliger Wert zwischen 60-200 für den gesamten Anstieg gewählt. Die Variable `x` wird mit 600 Werten zwischen null bis fünf befüllt. Dies dient dazu, um in der nächsten Zeile bei `growth` mit der Position auf der x-Achse rechnen zu können, was mit den normalen Frame-Indizes zu riesigen Werten führen würde.

Listing 22: Generierung einer riskanten Sequenz mit exponentiellem Anstieg

```
1       elif selected_pattern == "exponential_growth":
2               total_rise = np.random.uniform(60.0, 200.0)
3               x = np.linspace(0, 5, NUM_FRAMES, dtype=np.float64)
4               growth = ((np.exp(x) - 1) / (np.exp(5.0) - 1)) * total_rise
5               noise_std = np.random.uniform(RISK_NOISE_STD_LOW, 2.0)
6               noise = np.random.normal(0, noise_std, NUM_FRAMES)
7               temps = base + growth + noise
```

4.1.5 Modelltraining

Als Modell wurde ein LSTM (siehe Abschnitt 2.2.7) gewählt, welches mit der Bibliothek Pytorch (siehe Abschnitt 2.5.1) trainiert wurde.
Das LSTM wird mit den in Listing 23 gezeigten Parametern initialisiert. `input_dim` gibt die Anzahl der Features (siehe Abschnitt 2.2.5) pro Zeitschritt an, `hidden_dim` definiert die Größe eines Hidden Layer und `num_layers` gibt die Anzahl der Hidden Layers an.
Die Klassifikationsschicht `self.classifier` verarbeitet den Ausgabevektor des LSTMs zu einem Ergebnis. Der Vektor wird über drei Schichten von 64 Werten auf einen Ausgabe-Logit verdichtet. Weitere Informationen zu Neuronalen Netzen finden sich in Abschnitt 2.2.3. Die Funktion `forward` definiert, wie mit den Eingabedaten umgegangen wird. Sie werden zuerst dem LSTM übergeben, welches folgende Rückgabewerte anbietet: `output,(h_n,c_n)`. Der letzte Hidden State der obersten Schicht `h_n[-1]` des Modells wird an den `classifier` weitergegeben.

Listing 23: Aufbau des Modells

```
class RiskLSTM(nn.Module):
    def __init__(self, input_dim=1, hidden_dim=64, num_layers=2):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.classifier = nn.Sequential(
            nn.Linear(hidden_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )


    def forward(self, x):
        _, (h_n, _) = self.lstm(x)
        return self.classifier(h_n[-1]).squeeze()
```

Der mathematische Hintergrund und der Ablauf des Modelltrainings werden in den Abschnitten 2.2.3 und 2.2.7 bereits erläutert.

In Listing 24 wird das Modelltraining im Code veranschaulicht.

Durch `for epoch in range(EPOCHS)` wird die gewünschte Anzahl an Trainingsdurchläufen durchgeführt. `model.train()` versetzt das Modell in den Trainingsmodus. Dadurch werden z.B. zufällige Aktivierungsfunktionen auf null gesetzt, um zu verhindern, dass sich das Modell zu stark auf einzelne Neuronen konzentriert. `for batch_x, batch_y in train_loader` iteriert über zuvor gebildete Batches, welche dazu dienen, nicht alle Trainingsdaten auf einmal verarbeiten zu müssen. Dadurch wird Rücksicht auf den kleinen Arbeitsspeicher der Arbeitsgeräte der Projektmitglieder genommen. Mit `preds = model(batch_x).squeeze()` wird der Forward Pass durchgeführt. `loss.backward()` nimmt den Backward Pass vor. `optimizer.step()` passt die Gewichte an, um den Loss zu verringern.

Listing 24: Modelltraining - abstrakte Darstellung

```
for epoch in range(EPOCHS):
    model.train()
    for batch_x, batch_y in train_loader:
        preds = model(batch_x).squeeze()
        loss = loss_fn(preds, batch_y)
        loss.backward()
        optimizer.step()
```

4.1.6 Flask Server

Um das Backend nicht aufzublähen, wurde die Logik der Branderkennungskomponente in einen extra Flask-Server (siehe Abschnitt 2.4.8) ausgelagert. In Abbildung 44 ist der Aufbau des Servers zu sehen. In dem Unterordner Model ist das trainierte Modell bzw. dessen Gewichte gespeichert.

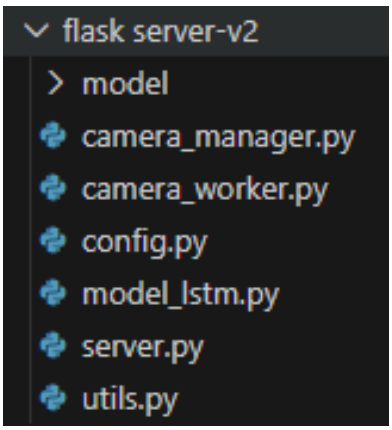


Abbildung 44: Aufbau Flask-Server

server

Der in Listing 25 präsentierte Code blockiert alle Anfragen von ausserhalb des Host-Rechners und prüft ob die Anfragen mit einem gültigen API-key versehen sind. Das Attribut `@app.before_request` ist ein sogenannter Flask-Decorator welcher bewirkt, dass dieser Code-Block jedes mal ausgeführt wird, bevor der Server einen Request bearbeitet [80].

Listing 25: Überprüfung eines Requests

```
@app.before_request
def check_api_key():
    if request.remote_addr not in ["127.0.0.1", "::1"]:
        return jsonify({"error": "Forbidden"}), 403

    key = request.headers.get("x-api-key")
    if key != API_KEY:
        return jsonify({"error": "Unauthorized"}), 401
```

In Listing 26 werden die zwei Endpunkte des Flask-Servers exemplarisch gezeigt. Diese werden im Backend (siehe Abschnitt 4.3.7) aufgerufen, wenn eine Kamera hinzugefügt bzw. gelöscht wird. Die Funktionen `register_camera` und `deregister_camera` sind im `camera_manager` (siehe Abschnitt 4.1.6) implementiert.

Listing 26: Endpunkte des Flask-Servers - abstrakte Darstellung

```
@app.route("/register_camera", methods=["POST"])
def register():
    register_camera(name, url, auth, camera_id)

@app.route("/unregister_camera", methods=["POST"])
def deregister():
    deregister_camera(name)
```

Das Startverhalten des Servers ist in Listing 27 illustriert. Bevor der eigentliche Flask-Server mit `app.run` gestartet wird, werden alle Kameras aus dem Backend geholt (siehe Abschnitt 4.3.7), um auch jene zu registrieren, die vor Starten des Servers bereits im Backend persistiert

wurden. Dieses Verhalten stellt einen Failsafe für ein unerwartetes Stoppen des Flask-Servers oder ein Nichteinhalten der Startreihenfolge der einzelnen Komponenten dar, da ansonsten bei einem Neustart des Servers alle Kameras im Backend manuell entfernt und wieder hinzugefügt werden müssten.

Listing 27: Startverhalten des Servers - abstrakt Darstellung

```
if __name__ == "__main__":
    def fetch_and_register_existing_cameras():
        try:
            url = f"{BACKEND_BASE_URL}/cameras"
            headers = {"Authorization": f"Bearer {BACKEND_API_KEY}"}
            resp = requests.get(url, headers=headers, timeout=5)
            if resp.status_code != 200:
                print(f"Failed to fetch cameras: {resp.status_code} {resp.text}")
                return

            cams = resp.json()

            ...

    fetch_and_register_existing_cameras()
    app.run(host="127.0.0.1", port=5001)
```

camera_manager

Der `camera_manager` ist eine relativ schlanke Klasse. Wie der Name bereits verrät, führt dieses Programm koordinative Tätigkeiten aus. Die Grundlage dafür bildet die Collection `camera_threads`. Wird eine neue Kamera über `register_camera` registriert, so wird ein neuer Thread (siehe Abschnitt 2.2.2) für diese gestartet. In `deregister_camera` wird überprüft, ob die übergebene id zu einem Eintrag in der Collection gehört. Ist dies der Fall, so wird das `stop_event` über `.set()` aktiviert und somit die Abbruchbedingung für die `run_camera_stream`-Funktion (siehe nächster Abschnitt) gesetzt. Informationen über Events bei Threads in Python sind in Abschnitt 2.4.8 zu finden.

Listing 28: Kamera Manager - abstrakte Darstellung

```
from camera_worker import run_camera_stream

camera_threads = {}

def register_camera(name, url, auth, camera_id, interval=1.0):
    thread = Thread(
        target=run_camera_stream,
        args=(name, url, auth, camera_id, interval, stop_event),
        daemon=True
        )
    camera_threads[key] = {"thread": thread, "stop_event": stop_event, "name": name}
    thread.start()

def deregister_camera(id: str) -> bool:
    key = str(id)
    if key in camera_threads:
        entry = camera_threads.pop(key)
        entry["stop_event"].set()
        return True
```

camera_worker

Der `camera_worker` ist zuständig für die Beschaffung der Temperaturdaten. Die erhaltenen Daten werden zuerst durch das trainierte Modell überprüft und daraufhin gemeinsam mit der Einschätzung des Modells an das Backend (siehe Abschnitt 4.3.8) zur Persistierung gesendet.
In Listing 29 ist die Zielfunktion des Threads zu sehen. Hier wird die, in Listing 28 festgelegte, Abbruchbedingung abgefragt. Ist diese gesetzt, so wird aus der Hauptschleife des Threads ausgebrochen und er wird somit beendet.
Die Funktion speichert eine Liste von Temperaturwerten in der Variable `values`. Wie dieser Wert aus dem Kommentarblock des JPEGs extrahiert wird, ist in Abschnitt 4.1.2 bereits beschrieben. Das Modell überprüft immer eine Spanne von 600 Datenpunkten. Dieser Wert ist durch die Konstante `SEQ_LEN`, aus der `config`-Datei (siehe nächster Abschnitt), festgelegt. Sind mehr als 600 Datenpunkte gesammelt, so wird der älteste mit `values.pop(0)` entfernt.

Listing 29: Kamera Worker Ausschnitt 1 - abstrakte Darstellung

```
1  def run_camera_stream(name, url, auth, camera_id, interval=INFERENCE_INTERVAL_SEC,
       stop_event):
2      values=[]
3
4      while True:
5          if stop_event.is_set():
6              break
7
8          try:
9              response = requests.get(stream_url, stream=True, auth=auth, timeout=5)
10
11             jpeg = extract_jpeg(response)
12             comment = extract_jpeg_comment(jpeg)
13             temp = extract_ttr_value(comment)
14
15             values.append(temp)
16             if len(values) > SEQ_LEN:
17                 values.pop(0)
```

Sind 600 Datenpunkte erreicht, so beginnt das Program die Daten auszuwerten. In `window` werden die derzeit betrachteten `values` als Numpy-Array (siehe Abschnitt 2.5.1) gespeichert und dabei gleich von Strings zu float-Werten konvertiert. Daraufhin wird die Funktion `predict_prob_risky()` (siehe Listing 31) mit `window` als Übergabewert aufgerufen. Übersteigt die Confidence des Modells den in `config` (siehe nächster Abschnitt) definierten Grenzwert, so wird `is_risky` auf True gesetzt. Anschließend wird der derzeitige Messwert gemeinsam mit der Einschätzung des Modells an das Backend (siehe Abschnitt 4.3.8) gesendet.

Listing 30: Kamera Worker Ausschnitt 2 - abstrakte Darstellung

```
def run_camera_stream(name, url, auth, camera_id, interval=INFERENCE_INTERVAL_SEC,
    stop_event):
    values=[]

    While True:

        ...

        if len(values) == SEQ_LEN:
            window = np.array(values, dtype=np.float32)
            p_risky = predict_prob_risky(window)
            is_risky = p_risky > THRESHOLD

        payload = {
                "time": now,
                "camera_id": camera_id,
                "temperature": float(parsed_temp),
                "confidence": round(p_risky, 4),
                "is_risky": bool(is_risky),
            }
        post_to_backend(payload, name)
```

In Listing 31 wird die Hilfsfunktion `predict_prob_risky()` dargestellt. Durch `.reshape(1,-1,1)` wird das Array an die, durch das Modell vorgegebene Form (siehe Abschnitt 2.2.3), angepasst. Für `-1` wird die Länge von `sequence` eingesetzt um das Muster `(batch size, time steps, features per step)` zu erfüllen. Aus den umgeformten Daten wird nun ein Pytorch Tensor (siehe Abschnitt 2.5.1) gebaut. `model(t)` gibt einen Zahlenwert zurück, der die Klassifizierung (siehe Abschnitt 2.2.4) der übergebenen Sequenz als riskant oder sicher repräsentiert. Ein negativer Rückgabewert bedeutet hierbei eine Einstufung als sicherer Verlauf und ein positiver bedeutet das Gegenteil. Da das Ergebnis von `model(t)` quasi keine Ober- bzw. Untergrenze hat (siehe Abschnitt 2.2.3) wird dieses nun noch durch `torch.sigmoid(logits)` zu einem Wert zwischen null und eins umgewandelt, um das Ergebnis in Prozent und somit in leichter lesbarer Form weitergeben zu können. [81]

Listing 31: Kamera Worker Ausschnitt 3 - abstrakte Darstellung

```
def predict_prob_risky(sequence: np.ndarray) -> float:
    x = sequence.astype(np.float32).reshape(1, -1, 1)
    t = torch.from_numpy(x)
    with torch.no_grad():
        logits = model(t)
        prob = torch.sigmoid(logits)
    return float(prob.item())
```

Durch die Zuweisung `model = _load_model()` wird `_load_model()` beim Import der `run_camera_stream`-Methode im `camera_manager` (siehe Listing 28) ausgeführt. Es wird zwar nur eine Methode von `camera_worker` importiert, jedoch lädt Python bei einem Import das ganze Programm und führt sogenannten 'Top-Level-Code' wie `model = _load_model()` direkt aus [82]. Mit `RiskLSTM()` wird das Modell initialisiert; die Struktur des Modells wird in Abschnitt 4.1.5 bereits beschrieben. Zu diesem Zeitpunkt sind die Gewichte noch mit Standartwerten befüllt.

Durch `torch.load(path, map_location=DEVICE)` wird das unter `/model` gespeicherte `.pt`-File welches die, beim Training ermittelten, Gewichte enthält geladen und de-serialisiert. Durch `map_location` ist es möglich, ein Modell, welches auf einer GPU trainiert wurde in der Produktion auf einer CPU laufen zu lassen und umgekehrt. `model.load_state_dict(state)` ersetzt die Gewichte im zuvor initialisierten Modell mit denen aus dem Training. Hierbei ist darauf zu achten, dass die Struktur des instanzierten Modells exakt der entsprechen muss, die bei dem Training angewandt wurde. Durch `model.to(DEVICE)` werden alle Modellparameter und Buffer auf das Zielgerät übertragen und `model.eval()` aktiviert den sogenannten Inferenzmodus. Dieser Modus deaktiviert Trainingsverhalten zum anpassen der Gewichte und versetzt das Modell in den Produktivzustand. [81]

Listing 32: Kamera Worker Ausschnitt 4 - abstrakte Darstellung

```
model = _load_model()

def _load_model() -> RiskLSTM:
    path = _resolve_model_path()
    model = RiskLSTM()
    state = torch.load(path, map_location=DEVICE)
    model.load_state_dict(state)
    model.to(DEVICE)
    model.eval()
    return model
```

config

In Listing 33 werden die globalen Einstellungen für den Flask-Server getroffen. Der `API_KEY` wird verwendet, um Anfragen des Backends zu validieren und `BACKEND_API_KEY` für Anfragen an das Backend. `SEQ_LEN` gibt an, wie viele Messwerte eine Sequenz beinhaltet. Multipliziert man diesen Wert mit der Frequenz in `INFERENCE_INTERVAL_SEC` so gewinnt man die Länge des beobachteten Zeitfensters in Sekunden. In der derzeitigen Konfiguration sind das 600 Sekunden, also 10 Minuten.

Listing 33: config-file

```
API_KEY = "*"
BACKEND_API_KEY = "*"
BACKEND_BASE_URL = "http://localhost:3000"

MODEL_PATH = os.getenv("MODEL_PATH", "model\\model_lstm-newV2.pt")
SEQ_LEN = int(os.getenv("SEQ_LEN", "600"))
INFERENCE_INTERVAL_SEC = float(os.getenv("INFERENCE_INTERVAL_SEC", "1.0"))
THRESHOLD = float(os.getenv("THRESHOLD", "0.5"))
POST_TO_BACKEND = os.getenv("POST_TO_BACKEND", "1") != "0"
```

4.2 Frontend

4.2.1 Komponentenaufbau

Die zentrale Einstiegskomponente des MoboView-Systems befindet sich im `'/src/app'` Verzeichnis. Sie beinhaltet lediglich das Router-Outlet (Siehe Abbildung 34), welches es bewerkstelligt, dass, wenn eine URL zu einer Route passt, Angular die zugeordnete Komponente erzeugt und deren View genau an der Stelle des Router-Outlets einhängt.

Listing 34: Variablen Import

```
<router-outlet></router-outlet>
```

Die eigentlichen Komponenten liegen unter `'src/app/features'`. Diese beinhalten wiederum jeweils ein TypeScript-File für die Logik, ein HTML-File für die Struktur und ein SCSS-File für das Styling (Siehe Abbildung 45).

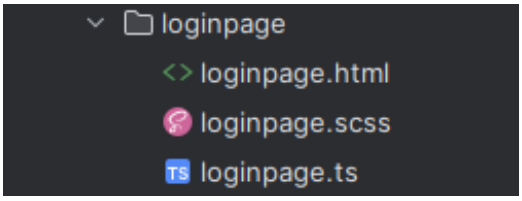


Abbildung 45: Komponentenstruktur

In `'src/app/features'` liegen also sozusagen alle Views, die gemäß dem Figma-Design (Siehe Abschnitt 2.1.1) umgesetzt wurden.

4.2.2 Routing und Navigation

Die zentrale Routen-Definition wird im `'app.routes'` TypeScript-File festgehalten (Siehe Listing 35). Jede, hier festgelegte URL zeigt direkt auf eine Komponente (View). Für jede Route existiert dann noch eine Liste an Rollen, die eine Zugriffsberechtigung auf diese Route haben.

Listing 35: Routen Definition

```
export const routes: Routes = [
  { path: '', component: Loginpage },
  { path: 'add-camera', component: AddCamera, canActivate: [canAlwaysActivateFn], data: {
      roles: ['admin', 'viewer', 'super_user'] } },
  { path: 'dashboard', component: Dashboard, canActivate: [canAlwaysActivateFn], data: {
      roles: ['admin', 'viewer', 'super_user'] } },
  { path: 'manage-users', component: ManageUsers, canActivate: [canAlwaysActivateFn],
      data: { roles: ['super_user'] }},
  { path: 'role',
```

Beim Aufruf der Login-Komponente wird geprüft, ob bereits ein Zugriffstoken im `LocalStorage` vorhanden ist. Ist dies der Fall, wird der Benutzer automatisch auf das Dashboard weitergeleitet (Siehe Listing 36). Dadurch wird vermieden, dass bereits authentifizierte Nutzer unnötig erneut auf der Login-Seite landen.

Listing 36: Routen Definition

```
ngOnInit(): void {
  const token = localStorage.getItem('accessToken');
  if (token) {
    this.router.navigate(['/dashboard']);
  }
}
```

Die eigentliche Navigation über das UI ist über Nav-Bars mittels Click-Handler-Mechanismus (Siehe Listing 37) abgeregelt.

Listing 37: Navbar Logik

```
handleOnTabClick(tab: string) {
    this.activeTab = tab;
    this.router.navigate(['/' + tab]);
}
```

Klickt man auf einen neuen Tab, wird `'activeTab'` aus der aktuellen URL abgeleitet, und dem aktiven Element wird eine SCSS-Klasse angehängt (Siehe Listing 38) um zu Veranschaulichen, welcher Tab nun ausgewählt ist (Siehe Abbildung 46).

Listing 38: Ausgewählter Tab

```
[ngClass]="{'activate' : activeTab === 'add-camera' }"
```

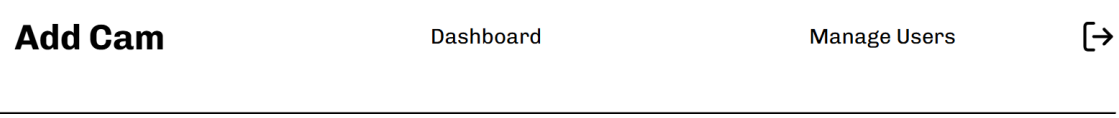


Abbildung 46: Ausgewählter Tab

In der Anwendung selbst existieren zwei unterschiedliche Navbars. Einerseits jene die der Viewer bzw. der Admin (Siehe Abbildung 47) und andererseits die, die der Super User angezeigt bekommt (Siehe Abbildung 48). Sie unterscheiden sich lediglich dadurch, dass die SuperUser-Navbar als dritten Tab keinen Rollen-Übersichts-Tab sondern stattdessen eine Manage-User Funktion beinhaltet.

Abbildung 47: Navbar Viewer & Admin

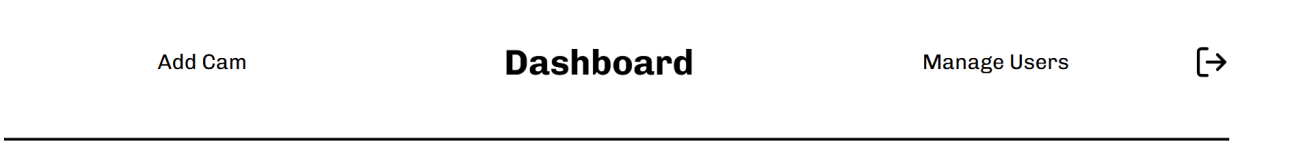


Abbildung 48: Navbar SuperUser

Die Einbindung der rollenspezifischen Navbar erfolgt dann folgendermaßen. Es werden beide Navbar-Komponenten in die jeweiligen Ansicht eingeschleust und dann wird mittels `'NgIf'`-Syntax abgefragt welche Rolle dem aktuellen User zugeteilt ist.

Listing 39: Einbindung Navbar

```
<app-navbar *ngIf="getRole() === 'admin' || getRole() === 'viewer'"></app-navbar>
<app-navbar-superuser *ngIf="getRole()=== 'super_user'"></app-navbar-superuser>
```

Dies wird dadurch ermöglicht, da bei der Initialisierung der Kernkomponente (Siehe Abschnitt 4.2.1) die Rolle des aktuellen Users aus der Datenbank ausgelesen und in einem Service zwischengespeichert wird (Siehe Listing 40).

Listing 40: Laden der User-Rolle

```
async ngOnInit(): Promise<void> {
    await this.role.loadUserRole();
}
```

Aber nicht nur bei der Rollen-spezifischen Navbar wurde auf die Verwendung der `'NgIf'`-Syntax gesetzt sonder auch bei der internen Navigation innerhalb einer Route. `'NgIf'` kam unter anderem in der Dashboard-Komponente, um zwischen Kamera-Feeds und Kamera-bezogener historischen Temperaturverläufe wechseln zu können (Siehe Abbildung 49).

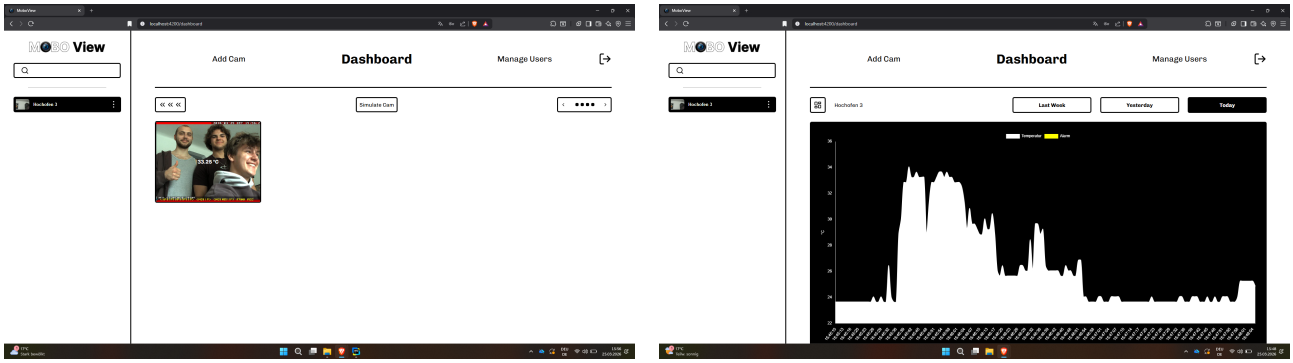


Abbildung 49: Dashboard

Für die Zugriffskontrolle wurde auf die Verwendung eines Guards gesetzt, denn nicht nur die Benutzeroberfläche soll durch eine Rollen-spezifische Navbar eingeschränkt werden, sondern es muss zusätzlich verhindert werden, dass geschützte Bereiche über direkte URL-Eingaben aufgerufen werden können. Der Guard entscheidet vor dem Navigieren, ob eine Route basierend auf der Rolle überhaupt geladen werden darf. Dies bewerkstelligt er indem er die Liste der berechtigten Rollen wie bereits obig erwähnt lädt und per `'.includes()'` überprüft, ob der User mit der aktuellen Rolle berechtigt ist, auf diese Route zuzugreifen (Siehe Listing 41).

Listing 41: Guard

```
const allowedRoles = route.data['roles'] as string[];

  if (!authService.getRole()) {
    await authService.loadUserRole();
  }

  const userRole = authService.getRole();

  if (!userRole || !allowedRoles.includes(userRole)) {
    router.navigate(['/']);
    return false;
  }

  return true;
```

Die Zugriffsberechtigungen je Rolle, wurde gemäß der Vorab-Überlegung implementiert (Siehe Abschnitt 2.1.1).

4.2.3 API-Anbindungen und Services

API-Anbindungen

Die API-Ressource `'app/core/api'` (Siehe Abbildung 50) bildet die konkreten Backend-Endpunkte (URLs, Parameter) ab und stellt die sinngemäße Kommunikation zwischen Frontend und Server bereit.

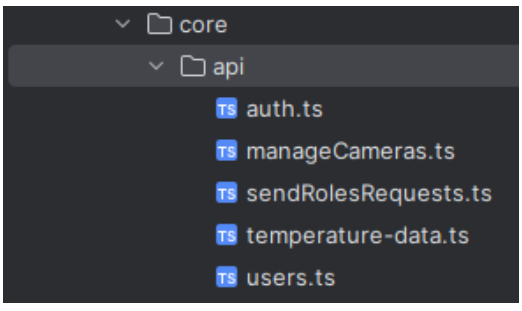


Abbildung 50: API-Schnittstellen

Für jede ansprechbare Ressource wurde ein eigenes TypeScript-File angelegt und darin die Basis-Route definiert. Dann wurden jeweils die verschiedenen Operationen, die auf diese Route

angewandt werden können mittels aussagekräftig beschlagworteten Methoden zugänglich gemacht (Siehe Listing 42).

Wichtig ist dabei, dass diese API-Klassen selbst in der Regel nur Endpunkt-Strings zurückgeben (String für URL-Pfade) und keine Requests ausführen. Der eigentliche HTTP-Aufruf erfolgt in den Komponenten.

Listing 42: Ausgewählter Tab

```
private readonly baseUrl = 'http://localhost:3000/api/role-requests';
  get roleRequest() {
    return this.baseUrl;
  }
  get getRoleRequest() {
    return `${this.baseUrl}/my-request`;
  }
  get allRequests() {
    return `${this.baseUrl}/pending`;
  }
  editRequest(id: number) {
    return `${this.baseUrl}/${id}`;
  }
```

Services

Die Services, welche sich unter `'app/core/services'` befinden, stellen die Vermittlungsschicht zwischen den UI-Komponenten und den Daten dar.

Folgende Services wurden implementiert.

Camera-Service

Das Camera-Service sorgt dafür, dass alle kamerabezogenen Bereiche in der Oberfläche immer sofort aktualisiert werden. Dafür stellt das Service zwei Observables bereit, Einerseits `camerasChanged&`, welches ein allgemeines Signal ist und bedeutet: „Bei den Kameras hat sich etwas geändert.“ Dieses Signal wird über den Methodenaufruf `'.triggerReload()'` ausgelöst und dient vor allem dazu, dass betroffene Komponenten ihre Daten neu laden. Andererseits gibt es `cameraAdded$`. Dieses Ereignis enthält die ID einer neu hinzugefügten Kamera und wird über `'.emitCameraAdded(id)'` gesendet (Siehe Listing 43). Im Zusammenspiel werden nach dem erfolgreichen Hinzufügen einer Kamera beide Signale verwendet. Zuerst `'.emitCameraAdded(id)'`, danach `'.triggerReload()'`. So wird zuerst die konkrete Kamera übermittelt und anschließend eine allgemeine Aktualisierung angeregt.

Listing 43: Camera Service

```
export class CameraService {
  private camerasChangedSubject = new Subject<void>();
  camerasChanged$ = this.camerasChangedSubject.asObservable();

  private cameraAddedSubject = new Subject<number>();
  cameraAdded$ = this.cameraAddedSubject.asObservable();

  triggerReload(): void {
    this.camerasChangedSubject.next();
```

```
10    }
11
12    emitCameraAdded(id: number): void {
13      this.cameraAddedSubject.next(id);
14    }
15  }
```

Role-Service

Die Aufgabe des Role-Services ist die Verwaltung der Benutzerrolle im Frontend. Es stellt dafür drei Methoden bereit. `'loadUserRole()'` übernimmt das Einlesen der Rolle aus dem Backend und speichert das Ergebnis direkt in das Rollen-Attribut. Mit `'getRole()'` kann dieser Wert anschließend im Code abgefragt werden. `'setRole(role)'` erlaubt zusätzlich das Überschreiben der Rolle (Siehe Listing 44).

Listing 44: Role Service

```
1  export class Role {
2    private role: string = '';
3
4    constructor(private http: HttpClient, private auth: Auth) {}
5
6    setRole(role: string): void {
7      this.role = role;
8    }
9
10   getRole(): string {
11     return this.role;
12   }
13
14   loadUserRole(): Promise<void> {
15     const token = localStorage.getItem('token');
16     if (!token) return Promise.resolve();
17
18     return this.http
19       .get<{ role: string }>(this.auth.userData, {
20         headers: { Authorization: `Bearer ${token}` }
21       })
22       .toPromise()
23       .then((res) => {
24         if (res != undefined) {
25           this.role = res.role;
26         }
27       })
28       .catch(() => {
29         this.role = '';
30       });
31   }
32 }
```

Temperature-Service

Das Temperature-Service bewerkstelligt es, dass die Temperaturverläufe auf dem aktuellen Stand bleiben. Dafür stellt das Service ein Daten-Observable `temperatureData$` zur Verfügung, welches immer den aktuellen Stand der historischen Temperaturdaten liefert (Siehe Listing 46).

Die Methode `'.setTemperatureData()'` übernimmt das Aktualisieren der Temperaturdaten und speichert die neuen Werte direkt im internen Subject (Siehe Listing 46).

Listing 45: Dashboard Temperature Service Einbindung

```
this.temperatureDataService.temperatureData$.subscribe((data) => {
      this.historicalChartData = data;
      console.log('[DASHBOARD] Chart data updated:', data.length, 'entries');
    });
```

Listing 46: Temperature Service

```
export class TemperatureDataService {
  private temperatureDataSubject = new BehaviorSubject<Temperature[]>([]);
  temperatureData$ = this.temperatureDataSubject.asObservable();

  setTemperatureData(data: Temperature[]): void {
    this.temperatureDataSubject.next(data);
  }

  getTemperatureData(): Temperature[] {
    return this.temperatureDataSubject.getValue();
  }
}
```

Encryption-Service

Das Encryption-Service verschlüsselt im Frontend sensible Daten (konkret das Kamera-Passwort) mit einem Public-Key und gibt das Ergebnis Base64-kodiert zurück. (Siehe Listing 47) Damit wird das Passwort vor dem Versand ans Backend geschützt.

An dieser Stelle wird nicht auf eine Einweg-Hashfunktion wie `SHA-256` gesetzt, weil in späterer Folge das Kamera-Passwort im System wieder im Klartext benötigt wird. Die KI-Komponente greift automatisiert auf die Mobotix-Kamera zu und muss dafür die Zugangsdaten entschlüsseln können, um Temperaturdaten auszulesen. Ein Hash wäre dafür ungeeignet, da er sich nicht zurückrechnen lässt. Deshalb wird an dieser Stelle auf asymmetrische Verschlüsselung gesetzt.

Listing 47: Encryption Service

```
export class EncryptionService {
  private readonly publicKey: forge.pki.rsa.PublicKey;

  constructor() {
    const PUBLIC_KEY_PEM = `-----BEGIN PUBLIC KEY-----
    ...
-----END PUBLIC KEY-----`;

    this.publicKey = forge.pki.publicKeyFromPem(PUBLIC_KEY_PEM);
  }

  encrypt(text: string): string {
    const encrypted = this.publicKey.encrypt(text, 'RSA-OAEP');
    return forge.util.encode64(encrypted);
  }
}
```

4.2.4 Formularlogik und Validierung

Benutzereingaben sind der Auslöser für Aktionen im UI. Deshalb wird bereits im Frontend geprüft, ob alle benötigten Felder befüllt sind und darauf Wert gelegt, dass die Bedienung intuitiv ist.

Login/Registrierung

Vor dem Absenden wird geprüft, ob Benutzername und Passwort gegeben sind. Fehlt eine der Eingaben, wird eine aussagekräftige Fehlermeldung geworfen (Siehe Listing 48).

Listing 48: Pflichtfeldpruefung

```
submitForm() {
  if (this.username !== '' && this.password !== '') {
    ...
  } else {
    this.error = 'Please enter a username and password';
  }
}
```

Add Camera

Beim Hinzufügen einer Kamera wird geprüft, ob alle Felder ausgefüllt sind (Siehe Listing 49). bzw. ob die eingegebene Kamera-URL ein gültiges Vorschaubild liefert. (Siehe Abschnitt 4.2.5)

Listing 49: Fehlerfeedback

```
if (this.stream_url == '' || this.camera_name == '' ||
    this.camera_username == '' || this.camera_pw_encrypted == '') {
  this.error = 'All fields are required';
} else if (this.imageLoadError) {
  this.error = 'Please enter a valid URL';
}
```

Bedienung per Enter-Taste

Für eine intuitivere Nutzung wird das Absenden aller Formulare nicht nur durch ein Klick-Event auf den Submit-Button ausgelöst, sondern auch durch ein Enter-Key-Pressed-Event (Siehe Listing 50).

Listing 50: Enter-Handling

```
private onEnterPressed(): void {
    ...
}
```

4.2.5 Funktionen

Wie bereits verbal beschrieben ist der Frontend-Part von MoboView eine Webapplikation, die zur Verwaltung und Überwachung von Mobotix-Kameras dient, und gleichzeitig eine frühzeitige

Branderkennung auf Kameraebene mit sich bringt. Die Hauptfunktionen des rollenbasierten UIs sind das Hinzufügen bzw. Verwalten von Kameras. Diese können von der globalen Liste auf das Dashboard verschoben werden. Die Kameras, welche dann im Dashboard in Form einer Kamera-Karte angezeigt werden, geben live sowie auch historische Temperaturdaten preis.

Kameras hinzufügen

Die Webapplikation erlaubt es, neue Kameras anzulegen (Siehe Listing 54). Dabei werden die Kamera-Parameter (Name, URL, Username, Password) eingegeben.

Bei der Kameraerfassung wird jede Eingabe an der Stream-URL sofort verarbeitet. Sobald sich das Feld ändert, wird automatisch ein Proxy-Call ausgelöst (Siehe Abschnitt 4.2.7), um ein Vorschaubild abzurufen (Siehe Listing 51). Erst wenn die Vorschau erfolgreich geladen wurde, wird das Absenden der Kamera freigeschaltet (Siehe Listing 52). Damit wird verhindert, dass nicht erreichbare Kameras gespeichert werden.

Listing 51: Eingabe triggert Vorschau

```
<input (input)="onInputChange($event)" [(ngModel)]="stream_url" name="url"
    class="input-field">
```

Listing 52: Vorschau-Erfolg schaltet Submit frei

```
async submitForm(): Promise<void> {
  if (this.stream_url !== '' && this.camera_name !== '' && this.camera_username !== '' &&
      this.camera_pw_encrypted !== '' && !this.imageLoadError) {
    // ...submit...
  } else if (this.imageLoadError) {
    this.error = 'Please enter a valid URL';
  }
}
```

Nach dem erfolgreichen Hinzufügen der Kamera wird diese dank der Subscription in der Kamera-Listen-Komponente auf das Subject, `'camerasChanged$'` welches im Kamera-Service gehalten wird (Siehe Abschnitt 4.2.3) sofort in der globalen Kameraliste sichtbar. Ab diesem Zeitpunkt startet ein Warm-up Countdown von zehn Minuten. Wird die Kamera in dieser Phase zu dem Dashboard hinzugefügt, wird nicht die Temperatur dargestellt, sondern der Countdown. (Siehe Listing 53) Dieser wurde sinngemäß implementiert, da der Flask Server zehn Minuten nach dem Zeitpunkt der Kamera-Registrierung noch keine Temperaturdaten klassifiziert werden können (Siehe Abschnitt 2.4.8).

Listing 53: Countdown im Dashboard

```
<div class="warmup-overlay" *ngIf="isWarmingUp(camera.camera_id)">
  <div class="warmup-countdown">
    {{ formatCountdown(warmupSeconds(camera.camera_id)) }}
  </div>
</div>

<div class="temperature-overlay" *ngIf="!isWarmingUp(camera.camera_id) &&
    newestTemperatures[camera.camera_id] as temp">
  <span [ngClass]="{
    'risky-temperature': temp.is_risky && camera.is_alarm_active,
    'normal-temperature': !temp.is_risky || !camera.is_alarm_active
  }">
    {{ temp.temperature }}  C
  </span>
</div>
```

Listing 54: Kamera Hinzufügen

```
this.http.post<Camera>(this.manageCameras.postCamera, body, {
  withCredentials: true
}).subscribe({
  next: response => {
    this.cameraService.emitCameraAdded(Number(response.id));
    this.cameraService.triggerReload();
  }
});
```

Kamera-Listen verwalten

In der Kameraliste kann der Benutzer Kameras filtern (1), umbenennen (2), ins Dashboard übernehmen (3) oder gegebenenfalls löschen (4).

(1) Filtern der Liste

Zweck: Schnelles Finden von Kameras anhand des Namens.

Technische Umsetzung: Die Eingabe wird direkt auf die lokale Liste angewendet (Siehe Listing 55), ohne Backend-Request.

Listing 55: Kamera-Liste filtern

```
this.filteredCameras = this.cameras.filter(camera => {
  return camera.camera_name.toLowerCase().includes(input.toLowerCase());
});
```

(2) Kamera umbenennen

Zweck: Änderung des Anzeigenamens einer Kamera.

Technische Umsetzung: Ein PATCH-Request aktualisiert den Namen am Backend, danach wird die Liste neu geladen (Siehe Listing 56).

Listing 56: Kamera umbenennen

```
this.http.patch<Camera>(this.manageCameras.putCamera(this.editingCamera?.id), body, {
  withCredentials: true
}).subscribe({
  next: () => {
    this.getCameras();
    this.disableEditing(this.editingInput);
  }
});
```

(3) Kamera ins Dashboard übernehmen

Zweck: Eine Kamera wird von der globalen Liste in die Live-Überwachung übernommen.

Technische Umsetzung: Per POST-Request wird die Kamera verlinkt (Siehe Listing 57), danach wird ein Reload über den Camera-Service angestoßen (Siehe Abschnitt 4.2.3).

Listing 57: Kamera ins Dashboard übernehmen

```
this.http.post(this.manageCameras.linkCamera, body, {
  withCredentials: true
})
.subscribe({
  next: () => {
    this.cameraService.triggerReload();
  }
});
```

(4) Kamera löschen

Zweck: Nicht mehr benötigte Kameras werden aus der Liste entfernt.

Technische Umsetzung: Ein DELETE-Request löscht die Kamera und die lokale Liste wird einfach aktualisiert (Siehe Listing 58).

Listing 58: Kamera löschen

```
this.http.delete(this.manageCameras.deleteCamera(camera.id), {})
.subscribe({
  next: () => {
    this.cameraService.triggerReload();
    this.filteredCameras = this.filteredCameras.filter(c => c.id !== camera.id);
  }
});
```

Dashboard-Funktionen

Das Dashboard ist die zentrale Live-Ansicht. Hier werden Kameras in Form von Karten dargestellt und laufend aktualisiert. Live- (1) bzw. historische Temperaturdaten (2) können pro Kamera angezeigt werden. Alarme können auf Kameraebene ausgeschaltet (3) bzw. die Linse gewechselt werden (4). Ebenso lassen sich die Kamera-Elemente wieder aus dem Dashboard entfernen (5).

(1) Live-Temperaturen abrufen

Zweck: Anzeige der aktuellsten Temperaturwerte pro Kamera.

Technische Umsetzung: Für jede Kamera wird der letzte Messwert über einen GET-Request geladen (Siehe Listing 59).

Listing 59: Live-Temperatur laden

```
this.http.get<TemperatureData>(url, {
  headers: { 'Content-Type': 'application/json' },
  withCredentials: true,
}).subscribe({
  next: (data) => {
    this.newestTemperatures[cameraId] = data;
  }
});
```

(2) Historische Daten anzeigen

Zweck: Analyse vergangener Temperaturverläufe für eine ausgewählte Kamera.

Technische Umsetzung: Beim Öffnen wird die Kamera geladen und die Historien-Ansicht aktiviert (Siehe Listing 60).

Listing 60: Historische Ansicht öffnen

```
this.http.get<Camera>(this.manageCameras.getCameraById(camera.camera_id), {
  withCredentials: true
}).subscribe((response) => {
  this.selectedHistoricalCameraName = response.camera_name ?? 'Unknown camera';
  this.selectedHistoricalCameraId = camera.camera_id;
  this.historicalData = true;
});
```

Daraufhin wird sich bei der Initialisierung der Chart-Komponente auf Änderungen der Temperaturdaten subscribed und definiert, dass bei jeder Änderung der Chart mit den neuen Daten gebaut wird. Logischerweise kommt diese reaktive Implementierung nur von Vorteil, wenn man sich unter dem Reiter `Today` befindet, da die beiden anderen Reiter `Yesterday` und `Last Week` bereits abgeschlossene Zeitfenster repräsentieren.

Die Einfärbung der Kurve bei riskanten Temperaturverläufen erfolgt dann indem die gesamte Messreihe in zwei Datensätze aufgeteilt wird: `Temperatur` (weiß) für normale Werte und `Alarm` (gelb) für kritische Werte. Bei einem Statuswechsel wird der aktuelle Punkt zusätzlich in die jeweils andere Serie übernommen, damit die Linie an der Übergangsstelle optisch durchgängig bleibt (Siehe Listing 61).

Listing 61: Einfärbung der Chart-Kurve

```
const safeValues: (number | null)[] = new Array(data.length).fill(null);
const riskyValues: (number | null)[] = new Array(data.length).fill(null);

for (let i = 0; i < data.length; i++) {
  const cur = data[i];
  const prev = i > 0 ? data[i - 1] : null;
  const changed = prev ? prev.is_risky !== cur.is_risky : false;

  if (cur.is_risky) {
    riskyValues[i] = cur.temperature;
    if (changed && prev && !prev.is_risky) {
      safeValues[i] = cur.temperature;
    }
  } else {
    safeValues[i] = cur.temperature;
    if (changed && prev && prev.is_risky) {
      riskyValues[i] = cur.temperature;
    }
  }
}

this.chart.data.datasets = [
  { label: 'Temperatur', data: safeValues, borderColor: 'white' },
  { label: 'Alarm', data: riskyValues, borderColor: 'yellow' }
];
```

(3) Alarmstatus umschalten

Zweck: Aktivieren bzw. Deaktivieren der Alarmfunktion pro Kamera.

Technische Umsetzung: Ein PATCH-Request aktualisiert den Alarmstatus am Backend und setzt ihn direkt im UI (Siehe Listing 62).

Listing 62: Alarmstatus umschalten

```
this.http.patch(
  'http://localhost:3000/api/users/me/alarm',
  { camera_id: id, is_alarm_active: nextState },
  { headers: { 'Content-Type': 'application/json' }, withCredentials: true }
).subscribe({
  next: () => { camera.is_alarm_active = nextState; }
});
```

(4) Linse wechseln

Zweck: Umschalten zwischen Normal- und Thermal-Linse pro Kamera, um je nach Situation die passende Ansicht zu nutzen (Siehe Listing 63).

Technische Umsetzung: Ein PATCH-Request setzt das neue Linsen-Attribut am Backend (Siehe Lising 63.

Listing 63: Linse umschalten

```
this.http.patch<{ camera_id: number; camera_lens: CameraLens }>(
  this.manageCameras.patchLens(id),
  { camera_lens: next },
  {
    headers: { 'Content-Type': 'application/json' },
    withCredentials: true,
  }
).subscribe({
  next: (resp) => {
    ...
  }
});
```

(5) Kamera aus dem Dashboard entfernen

Zweck: Eine Kamera wird aus der Live-Überwachung entfernt.

Technische Umsetzung: Ein DELETE-Request entfernt die Verlinkung, anschließend wird die Liste aktualisiert (Siehe Listing 64).

Listing 64: Kamera vom Dashboard entfernen

```
this.http.delete(this.manageCameras.deleteLink(camera.camera_id), {
  withCredentials: true,
}).subscribe({
  next: () => {
    this.cameras = this.cameras.filter(c => c.camera_id !== camera.camera_id);
    this.cameraService.triggerReload();
  }
});
```

Elemente Selektor

Der Elemente Selektor steuert, wie viele Kamerakarten pro Reihe angezeigt werden. Die im UI dargestellten Pfeile rufen per Klick-Event `'.decrease()'` bzw. `'.increase()'` auf und verändern die Variable `'currentCount'`. Die Punkte in der Mitte werden dynamisch mittels `'ngFor'`-Syntax gerendert, und das Layout wird per `'[ngClass]'` je nach currentCount auf dashboard, dashboard4 oder dashboard5 umgeschaltet (Siehe Listing 65).

Listing 65: Elemente Selektor

```
<div class="elements-selector">
  <img (click)="decrease()" class="nav-arrow"
       src="resource/icons/arrow_left_elements_in_row.svg" alt="arrow-left">

  <div class="elements-svg-container">
    <ng-container *ngFor="let item of [].constructor(currentCount)">
      <svg width="16" height="16" viewBox="0 0 10 10">
        <rect x="3" y="3" width="6" height="6" rx="2" fill="black"/>
      </svg>
    </ng-container>
  </div>

  <img (click)="increase()" class="nav-arrow"
       src="resource/icons/arrow_right_elements_in_row.svg" alt="arrow-right">
</div>

<div [ngClass]="{
  'dashboard5' : currentCount === 5,
  'dashboard4' : currentCount === 4,
  'dashboard'  : currentCount === 3
}"></div>
```

Rollenanfrage

Zweck: Ein Benutzer beantragt eine Admin-Rolle.

Technische Umsetzung: POST-Request an den Role-Requests-Endpoint (Siehe Listing 66).

Listing 66: Rollenanfrage stellen

```
const body = { requested_role: 'admin' };
this.http.post(this.sendRoleRequest.roleRequest, body, {
  withCredentials: true
}).subscribe({
  next: () => this.checkPendingRequest(),
  error: (error) => console.error('Anfrage fehlgeschlagen:', error)
});
```

Benutzer- und Rollenverwaltung

Der Super User besitzt ein Verwaltungsrecht, um Benutzerrollen zentral zu ändern (1) bzw. Rollen-Anfragen akzeptieren (2) zu können.

(1) Benutzerrolle ändern

Zweck: Änderung einer Rolle.

Technische Umsetzung: PUT-Request an den User-Endpoint (Siehe Listing 67).

Listing 67: User-Rolle ändern

```
this.http.put<any>(this.userApi.setUserRole(Number(user.id)), body, {
  withCredentials: true
}).subscribe((response) => {
  user.role = response.role;
});
```

(2) Rollen-Anfrage bearbeiten *Zweck:* Annehmen oder Ablehnen von Rollen-Anfragen. *Technische Umsetzung:* PATCH-Request an die Request-Ressource, anschließend werden die Anfragen neu gefiltert (Siehe Listing 68).

Listing 68: Rollen-Anfrage bearbeiten

```
this.http.patch(this.roleRequests.editRequest(Number(request.id)), bodyRequest, {
  withCredentials: true
}).subscribe(() => {
  this.requests = this.requests.filter(r => r.id !== request.id);
});
```

4.2.6 Fehlerbehandlung

Die Fehlerbehandlung in MoboView ist konsistent umgesetzt. Entweder entsteht ein Fehler im Backend (Siehe Listing 69), oder er wird bereits im Frontend erkannt (Siehe Listing 70). In beiden Fällen wird in der jeweiligen Komponente ein Fehlerzustand gesetzt, der anschließend im HTML sichtbar gemacht wird (Siehe Listing 71).

Listing 69: Backend-Fehler behandeln

```
this.http.post<Camera>(this.manageCameras.postCamera, body, {
  withCredentials: true
}).subscribe({
  next: response => {
    this.error = '';
  },
  error: error => {
    this.error = error.error?.error || 'Serverfehler';
  }
});
```

Listing 70: Frontend-Fehler setzen

```
if (this.stream_url == '' || this.camera_name == '' ||
    this.camera_username == '' || this.camera_pw_encrypted == '') {
  this.error = 'All fields are required';
} else if (this.imageLoadError) {
  this.error = 'Please enter a valid URL';
}
```

Listing 71: Fehlermeldung im Template

```
<div *ngIf="error" class="error-message">
  {{ error }}
</div>
```

4.2.7 Kamera-Feed Proxy

Der Zugriff auf den Live-Feed der Kameras erfolgt nicht direkt aus dem Browser, sondern über einen zwischengeschalteten Kamera-Feed-Proxy, welcher in Python mit Flask implementiert ist. Er stellt einen HTTP-Endpunkt unter **'https://localhost:5000/proxy/'** zur Verfügung, welcher eingehende Anfragen vom Frontend entgegennimmt, dann an die Kamera weiterleitet und schließlich die Antwort an den Browser zurückgibt. Der Proxy übernimmt im Prozess die Authentifizierung gegenüber der Kamera (Siehe Listing 72) mittels Digest Authentication und setzt die HTTP-Header so, dass der Browser die Antwort ohne Verletzung der Same-Origin- bzw. CORS-Richtlinien verarbeiten kann (Siehe Listing 73).

Listing 72: Authentifizierung

```
auth = HTTPDigestAuth(CAM_USER, CAM_PASS)
```

Listing 73: Kamera-Feed Proxy

```
@app.after_request
def apply_cors_headers(response):
  response.headers["Access-Control-Allow-Origin"]  = "*"
  response.headers["Access-Control-Allow-Headers"] = "Content-Type, Authorization"
  response.headers["Access-Control-Allow-Methods"] = "GET, POST, OPTIONS"
  return response
```

Es wird unter anderem der Header `Access-Control-Allow-Origin` gesetzt, der dem Browser explizit erlaubt, Antworten dieses Proxy-Endpunkts von einem anderen Origin, in diesem Fall der

Angular-Anwendung auf `https://localhost:4200`, auszulesen. Ergänzend dazu definiert der Proxy über die Header `Access-Control-Allow-Methods` und `Access-Control-Allow-Headers`, welche HTTP-Methoden und welche benutzerdefinierten Header, z.B. `Content-Type` vom Browser gesendet werden dürfen.

Zu Beginn wurde versucht, den Kamera-Feed direkt im Frontend über ein gewöhnliches `<img>`-Tag einzubinden, indem die URL der Kamera direkt als Bildquelle angegeben wurde. Dieser Ansatz scheiterte jedoch an CORS-bedingten Einschränkungen und der Handhabung des Mitsenden der Zugangsdaten. Infolgedessen wurden die Authentifizierung, die eigentliche Kamerakommunikation sowie die Anpassung der CORS-Header wie obig beschrieben in einen Kamera-Feed Proxy ausgelagert.

Der Proxy wird im Angular (Siehe Abschnitt 2.4.7) Frontend an drei Stellen eingesetzt, um Kamera-Streams und Vorschauen zu gewährleisten.

(1) Dashboard

Zweck: Anzeige des Live-Kamera-Feeds pro Kamera.

Technische Umsetzung: Die Source wird über die Stream-URL aufgebaut und via Proxy geladen (Siehe Listing 74).

Listing 74: Dashboard Live-Feed via Proxy

```
1  [src]="'http://localhost:5000/proxy/control/faststream.jpg?target_ip=' +
       getStreamUrl(camera) + '&stream=full&fps=16&rand=214160'"
```

(2) Camera List

Zweck: Anzeige eines statischen Vorschaubildes je Kamera

Technische Umsetzung: Das Vorschaubild wird mit der Kamera-Stream-URL über den Proxy aufgerufen (Siehe Listing 75).

Listing 75: Camera List Vorschau via Proxy

```
1  src="http://localhost:5000/proxy/cgi-bin/image.jpg?&target_ip={{camera.stream_url}}"
```

(3) Add Camera

Zweck: Laden eines Vorschaubildes, welches anschließend gecroppt wird.

Technische Umsetzung: Beim Eingeben der Stream-URL wird ein Bild geladen (Siehe Listing 76), gecropped und in dem Platzhalter-Element angezeigt wird.

Listing 76: Add Camera Vorschau via Proxy

```
1  this.image =  http ://localhost:5000/proxy/cgi-bin/image.jpg&target_ip=${this.stream_url
```

4.3 Backend

4.3.1 app.ts - Application Entry Point

Überblick

Die Datei `app.ts` ist der Haupteinstiegspunkt der Express-Applikation und verwaltet die Initialisierung des gesamten Backends. Sie ist verantwortlich für die Einrichtung der Datenbankverbindungen (MSSQL (2.4.5) und TimescaleDB (2.4.6) und das Initialisieren der Tabellen, das Laden der Umgebungsvariablen über `.env`, die Konfiguration von Middleware, die Einbindung von Swagger/OpenAPI-Dokumentation (siehe 2.5.2 und 2.4.4) und das Mounting aller API-Router (siehe 2.4.3). Zusätzlich kümmert sich `app.ts` um geplante Aufräumaufgaben durch Cron-Jobs (siehe 2.5.2), die automatisch verwaiste Daten löschen und Konsistenzprüfungen durchführen. Die Datei orchestriert damit den gesamten Startup-Prozess und sorgt dafür, dass alle Komponenten des Backends in der richtigen Reihenfolge initialisiert werden. Das Backend kann mittels `npm run backend` gestartet werden, wo es durch `nodemon` [83] jede Dateiänderung überwacht und dementsprechend den Server neustartet. **Außerdem wurde während der gesamten Implementierung besonderer Wert auf ein klares und strukturiertes Error-Handling mit aussagekräftigen Fehlermeldungen und Statuscodes gelegt.**

Imports und Express-Initialisierung

Am Anfang werden alle notwendigen Module und Router importiert (77). Das Modul `dotenv/config` wird als erstes importiert, um alle Umgebungsvariablen aus der `.env`-Datei zu laden (siehe 2.1.6). Die Datenbankverbindungen zu MSSQL und TimescaleDB, alle vier Router (`user`, `camera`, `role-request`, `temperature-data`, (siehe 4.3.4, 4.3.7, 4.3.5, 4.3.8 und 4.3.6)), die Cleanup-Funktionen und die Cron-Bibliothek werden geladen. Anschließend wird die Express-App mit `express()` erstellt und der Server-Port auf `3000` gesetzt.

Listing 77: Imports

```
1  import 'dotenv/config';
2  import express from 'express';
3  import * as connectMSSQLDB from './database/connections/db-connection.mssql';
4  import { userRouter } from "./backend/router/user.router";
5  import { cameraRouter } from "./backend/router/camera.router";
6  import roleRequestRouter from "./backend/router/role-request.router";
7  import temperatureDataRouter from "./backend/router/temperature-data.router";
8  import { runMSSqlScript } from "./database/scripts/runMSSqlScript";
9  import cron from 'node-cron';
10 import { connectTimescaleDB } from './database/connections/db-connection.timescaledb';
11 import { runTimescaleSqlScript } from './database/scripts/runTimescaleSqlScript';
12 import { getPool as getMSSQLPool } from './database/connections/db-connection.mssql';
13 import { deleteOldTemperatureData } from
       './backend/repository/temperature-data.repository';
14 import { deleteCameraCompletely } from './backend/repository/camera.repository';
15 import { runCameraTemperatureConsistencyCheck } from
       './backend/service/consistency.service';
```

Swagger-Konfiguration

Die Swagger-Konfiguration (siehe 4.3.2) verwendet `swagger-jsdoc`, um JSDoc-Annotations aus den Router-Dateien zu sammeln und ein OpenAPI 3.0.0-Spezifikationsdokument zu generieren (siehe 2.4.4 und 2.5.2). Die Dokumentation wird unter dem Endpoint `/api-docs` bereitgestellt und ermöglicht interaktives Testen der API-Endpoints direkt im Browser. In Produktion wird diese später deaktiviert, um die API-Struktur nicht öffentlich zu machen. Es wird auch CORS (siehe 2.5.2) konfiguriert, welches nur `localhost` auf dem Port 4200 erlaubt, der Adresse des Frontends, um Anfragen von bösartigen Websites zu verhindern.

Listing 78: Initialisieren und Konfigurieren von Swagger/Middleware/Root-Route label

```
let cors = require('cors');

// Swagger/OpenAPI
import swaggerUi from 'swagger-ui-express';
const swaggerJsdoc: any = require('swagger-jsdoc');

// create express application
const app = express();
const port = 3000;

const allowedOrigins = [
    "http://localhost:4200"
];

// mount middleware
app.use(cors({
    origin: (origin: string | undefined, callback: (err: Error | null, allow?: boolean) =>
        void) => {
        if (!origin || allowedOrigins.includes(origin)) {
            callback(null, true);
        } else {
            callback(new Error('Not allowed by CORS'));
        }
    },
    credentials: true
}));

// Parse incoming HTTP request bodies as JSON
app.use(express.json());

// Swagger setup: generate spec from JSDoc annotations in backend/router
const swaggerOptions = {
    definition: {
        openapi: '3.0.0',
        info: {
            title: 'FireDetectionApp API',
            version: '1.0.0',
            description: 'API documentation for FireDetectionApp'
        },
        servers: [
            { url: `http://localhost:${port}`, description: 'Local server' }
        ],
        components: {
            securitySchemes: {
                bearerAuth: {
                    type: 'http',
                    scheme: 'bearer',
                    bearerFormat: 'JWT'
                },
                ApiKeyAuth: {
                    type: 'apiKey',
                    in: 'header',
                    name: 'x-api-key'
                }
            }
        }
    },
    apis: ['./backend/router/**/*.ts']
};
```

```
59
60 // Only mount Swagger UI when explicitly enabled via env var SWAGGER_ENABLED='true'
61 if (process.env.SWAGGER_ENABLED) {
62     const swaggerSpec = swaggerJsdoc(swaggerOptions as any);
63     app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));
64     console.log('Swagger UI mounted at /api-docs (SWAGGER_ENABLED=true)');
65 } else {
66     console.log('Swagger UI is disabled. Set SWAGGER_ENABLED=true to enable it.');
67 }
68
69 // endpoint for root route
70 app.get('/', (req, res) => {
71     res.send('Welcome to the MOBOView backend/api!');
72 });
```

Listing 79: Mounten der Router

```
1 // mount router(s)
2 app.use("/api/users", userRouter);
3 app.use("/api/cameras", cameraRouter);
4 app.use("/api/role-requests", roleRequestRouter);
5 app.use("/api/temperature-data", temperatureDataRouter);
```

Init-Funktion

Die `init()`-Funktion ist das Herzstück der Server-Startsequenz und wird am Ende der Datei aufgerufen. Sie verbindet sich zuerst mit der MSSQL-Datenbank und führt das `mssqldb-setup.sql`-Skript aus, das alle Tabellen, Indizes und Constraints erstellt, falls sie nicht existieren. Danach werden Testdaten und ein Superuser über SQL-Skripte eingefügt. Fehler bei diesen optionalen Schritten werden geloggt, aber stoppen den Server nicht. Anschließend wird die Verbindung zu TimescaleDB aufgebaut und das `timescaledb-setup.sql`-Skript ausgeführt (siehe 2.1.5). Nach erfolgreicher Verbindung zu beiden Datenbanken wird der HTTP-Server auf Port 3000 gestartet. Danach werden sofort Cleanup-Jobs ausgeführt, um vorhandene inkonsistente Daten aufzuräumen (siehe 2.5.2 und 2.1.6).

Listing 80: Initialisierungsfunktion von app.ts

```
1  init();
2
3  async function init() {
4      try {
5          // connect to MSSQL
6          await connectMSSQLDB.connect();
7          console.log('\nConnected to MSSQL DB.');
8          await runMSSqlScript('./database/scripts/mssqldb-setup.sql');
9          console.log('MSSQL Database schema initialized.');
10         try {
11             await runMSSqlScript('./database/scripts/testdata.sql');
12             console.log('MSSQL test data inserted!');
13         } catch (error) {
14             if (error instanceof Error) {
15                 console.error('Error inserting test data (MSSQL):', error.message);
16             } else {
17                 console.error('Error inserting test data (MSSQL):', error);
18             }
19             console.log('Server will still start - test data (MSSQL) will be skipped.');
20         }
21         try {
22             await runMSSqlScript('./database/scripts/setup-superuser.sql');
23             console.log('Super user (MSSQL) created!');
24         } catch (error) {
25             if (error instanceof Error) {
26                 console.error('Error creating super user (MSSQL):', error.message);
```

```
27              }
28          }
29
30          // connect to TimescaleDB
31          try {
32              await connectTimescaleDB();
33              console.log('Connected to TimescaleDB successfully.');
34              await runTimescaleSqlScript('./database/scripts/timescaledb-setup.sql');
35              console.log('TimescaleDB setup script executed successfully.');
36          } catch (error) {
37              console.error('Error during TimescaleDB initialization:', error);
38          }
39
40          // Start HTTP server
41          app.listen(port, () => {
42              console.log(`\n\nServer listening on port ${port}.\n\n`);
43          });
```

Listing 81: Cleanup-Job direkt nach Start

```
1   (async () => {
2               try {
3                   console.log('[STARTUP] Running initial cleanup tasks...');
4                   await cleanupSessions();
5                   const autoFix = (process.env.CONSISTENCY_AUTOFIX || 'false').toLowerCase()
                        === 'true';
6                   const report = await runCameraTemperatureConsistencyCheck(autoFix);
7                   console.log(`[STARTUP] Consistency check:
                        orphans=${report.orphanCameraIds.length},
                        deletedRows=${report.deletedRows}`);
8                   const deletedTemp = await deleteOldTemperatureData(7);
9                   console.log(`[STARTUP] Old temperature data deleted: ${deletedTemp}
                        entries removed.`);
10                  await cleanupCameras();
11                  console.log('[STARTUP] Initial camera cleanup completed.');
12              } catch (err) {
13                  console.error('[STARTUP] Error during initial cleanup tasks:', err);
14              }
15          })();
16      } catch (error) {
17          console.error('Critical error during server initialization:', error);
18          process.exit(1); // End the process with an error code
19      }
```

Cleanup-Jobs

Um zu verhindern, dass die Datenbanken mit verwaisten oder abgelaufenen Daten befüllt werden, gibt es mehrere automatisierte Cleanup-Funktionen. `cleanupSessions()` entfernt abgelaufene Authentifizierungssessions, `cleanupCameras()` löscht Kameras ohne Benutzer-Zuordnung inklusive ihrer Temperaturdaten, und `runCameraTemperatureConsistencyCheck()` identifiziert und löscht, je nach `CONSISTENCY_AUTOFIX`-Flag (siehe 2.1.6) verwaiste Temperaturmessdaten. Diese Aufgaben werden durch geplante Cron-Jobs (2.5.2) automatisiert. Täglich um 03:00 Uhr läuft der Session-Cleanup mit Konsistenzprüfung, um 03:01 Uhr werden alle Temperaturmessungen gelöscht, die älter als eine Woche sind, und jährlich am 1. Januar um 04:00 Uhr werden ungenutzte Kameras entfernt. Dies stellt sicher, dass die Datenbanken performant und konsistent bleiben.

Listing 82: Beispiel: Cleanup-Funktion zum Aufräumen abgelaufener Sessions

```
async function cleanupSessions() {
    try {
        // Delete sessions that have expired
        const pool = getMSSQLPool();
        if (!pool) throw new Error('MSSQL pool not initialized!');
        await pool.request().query(`DELETE FROM sessions WHERE expires_at <
            SYSDATETIMEOFFSET();`);
        console.log('Daily session cleanup completed.');
    } catch (err) {
        if (err instanceof Error) {
            console.error('Error during session cleanup:', err.message);
        } else {
            console.error('Error during session cleanup:', err);
        }
    }
}
```

Listing 83: Beispiel: Jährlicher Cron Job zum Aufräumen unverlinkter Kameras

```
// Yearly cron job: Cleanup cameras not associated with any user (every January 1st at
    04:00)
cron.schedule('0 4 1 1 *', async () => {
    try {
        await cleanupCameras();
        console.log('[CRON] Yearly camera cleanup completed.');
    } catch (err) {
        console.error('[CRON] Error during camera cleanup:', err);
    }
});
```

Die Syntax der Zeitangabe von Cron Jobs setzt sich aus fünf Ziffern zusammen, wobei die erste für die Minutenangabe steht, die zweite für Stunden, der dritte Parameter steht für den Monatstag, der vierte für den Monat und der letzte für den Wochentag. Somit bedeutet `'0 4 1 1 *'` jährlich am 1.1. um 04:00 Uhr. [69]

consistency.service.ts

`consistency.service.ts` implementiert eine Konsistenzprüfung zwischen MSSQL und TimescaleDB, um sicherzustellen, dass keine verwaisten Temperaturmessungen in der Datenbank existieren. Die zentrale Funktion `runCameraTemperatureConsistencyCheck()` arbeitet in drei Schritten. Zuerst werden alle `camera_ids` aus der TimescaleDB-Tabelle `temperature_data` geladen, deren Kamera nicht mehr existiert, dann werden alle Kamera-IDs aus der MSSQL-Tabelle `cameras` abgerufen, und anschließend werden die Differenzen berechnet, also Camera-IDs, die in TimescaleDB vorhanden sind, aber nicht mehr in MSSQL existieren. Diese verwaisten IDs zeigen an, dass eine Kamera gelöscht wurde, aber ihre Messdaten nicht mitgelöscht wurden.

4.3.2 OpenAPI Dokumentation

Die OpenAPI-Dokumentation wird durch JSDoc-Kommentare mit `@openapi`-Tags direkt am Anfang jeder Router-Datei aufgebaut und besteht aus drei Hauptebenen. Auf der obersten Ebene werden globale `tags` definiert, z.B. `tags: - name: Users`, die alle Endpoints

einer Kategorie gruppieren, und `components` mit allen wiederverwendbaren Schemata wie `User, Camera, UserCamera` und deren Feld-Definitionen mit Typ, erforderliche Felder und Beschreibungen. Diese Schemata können dann in den Endpoints via beispielsweise `$ref: '#/components/schemas/User'` referenziert werden. Auf der zweiten Ebene werden die `paths` definiert, die die API-Struktur abbilden. Somit wird für jeden Endpoint (z.B. `GET /api/users, POST /api/cameras/create`) separat dokumentiert, welche HTTP-Methode verwendet wird, welche `tags` zugeordnet sind, ein `summary`, und optional ein `requestBody` mit erforderlichen Feldern und deren Schema. Auf der dritten Ebene werden die responses dokumentiert, für jeden HTTP-Status-Code wird eine Beschreibung und optional das Response-Schema angegeben, z.B. `'200': description: Array of users, schema: type: array, items: $ref: User`. Zusätzlich wird die `security` pro Endpoint definiert, da manche `bearerAuth: []`, also ein JWT-Token im Authorization-Header erfordern, andere ApiKeyAuth für interne Services, und wieder andere haben keine Security-Anforderung, sie sind also öffentlich. Die Dokumentation folgt dabei dem OpenAPI 3.0.0-Standard und wird von swagger-jsdoc am Server-Startup automatisch geparst und in ein einzelnes OpenAPI-Dokument zusammengeführt, das dann unter `/api-docs` als Swagger UI verfügbar gemacht wird, damit beispielsweise Frontend-Entwickler die gesamte API-Struktur interaktiv erkunden können, ohne den Quellcode lesen zu müssen. [2.4.4] [2.5.2]

4.3.3 Middleware

Überblick über die Middleware

Das Middleware-System bildet die zweite Sicherheitsschicht des Backends nach der JWT-Konfiguration (siehe 4.3.4) und ist verantwortlich für die Validierung, Authentifizierung und Autorisierung aller eingehenden Anfragen. Sie werden in einer festgelegten Reihenfolge in den Router-Definitionen angewendet und filtern Anfragen aus, die bestimmte Kriterien nicht erfüllen. Da sie in der Express-Middleware-Pipeline durchlaufen werden, können sie Anfragen stoppen, wenn Kriterien nicht erfüllt sind.

authenticateToken

Die `authenticateToken()`-Middleware ist die wichtigste Komponente der Authentifizierung und wird auf fast allen Endpoints angewendet.

Workflow: Der Token wird aus dem `Authorization`-Header extrahiert, wobei die `authenticateToken()` das Format `Authorization: Bearer <token>` oder einen rohen Token akzeptiert. Sollte kein Token vorhanden sein, bekommt der Anfragende einen `StatusCodes.UNAUTHORIZED`-Error. Nachdem der Token erfolgreich extrahiert wurde, wird die dazupassende Session aus der Datenbank geladen. Sollte es keinen passenden Session-Eintrag geben, wird `StatusCodes.FORBIDDEN` retourniert. Wird jedoch ein Eintrag gefunden, überprüft die Middleware, ob dieser noch gültig ist, da ansonsten die Session als 'abgelaufen' zählt, sie aus der Datenbank gelöscht und die Anfrage abgelehnt wird. Anschließend wird mittels `jwt.verify()` die Signatur des Tokens überprüft. Insofern diese gültig und der Algorithmus 'HS256' ist, wird der Access Token dekodiert und die `user.id` extrahiert (siehe 4.3.4 und 4.3.5). Mit diesem Attribut kann der User aus der Datenbank geladen, in `req.user` gespeichert und allen nachfolgenden Middlewares zur Verfügung gestellt werden.

authorizeRole

`authorizeRole()` implementiert ein rollenbasiertes Zugriffssystem (siehe 2.1.4), welches direkt nach der erfolgreichen Authentifizierung des Benutzers zum Einsatz kommt. Dabei können dieser Middleware ein oder mehrere Parameter übergeben werden, z.B. `authorizeRole('admin', 'super_user')` für eine Route, die nur Admins und der Super User aufrufen darf. Es wurde auch die vorkonfigurierte Middleware-Instanz `requireSuperuser` erstellt, welche nur den Super User erlaubt, da das bei mehreren Routen der Fall ist und somit der Code besser lesbar, sowie die Middleware einfacher zu implementieren ist.

Workflow: Es wird geprüft, ob die `req.user.role` in der Liste mit den erlaubten Rollen vorhanden ist, woraufhin `next()` aufgerufen und das Objekt an die nächste Middleware oder Handler weitergegeben wird, wenn seine Rolle beim aufgerufenen Endpunkt erlaubt ist. Andernfalls wird der Client darüber informiert, dass der Zugriff verweigert und seine Rolle nicht akzeptiert wurde bzw. unausreichend ist.

Listing 84: authorizeRole-Workflow

```
    return (req: Request, res: Response, next: NextFunction): void => {
        // --- 1) Ensure authentication produced a user with a role ---
        if (!req.user?.role) {
            console.error('Role check failed: No role present.');
            res.status(StatusCodes.UNAUTHORIZED).json({ error: 'No role present.' });
            return;
        }

        // --- 2) Verify role is allowed ---
        if (!allowedRoles.includes(req.user.role)) {
            console.error(`Role check failed: Access denied for role ${req.user.role}.`);
            res.status(StatusCodes.FORBIDDEN).json({ error: 'Access denied: insufficient
                permissions' });
            return;
        }

        // Role allowed -> continue
        next();
    };
}
```

Listing 85: Beispiel-Route: Nur Admin und Super User erlaubt

```
cameraRouter.post('/create', authenticateToken, authorizeRole('admin', 'super_user'),
    validateBody, async (req: Request, res: Response)
```

validateBody

`validateBody()` überprüft, ob ein Request-Body vorhanden ist, bevor er von den Routern verarbeitet wird, um leere oder fehlende Request-Bodies und die darauffolgenden Fehler zu vermeiden. Die Middleware wird auf `POST`, `PUT` und `PATCH` angewendet. Sollte der Body `null` oder `undefined` sein, bekommt der Client eine passende Fehlermeldung, wohingegen bei einem vorhandenen Body wieder `next()` aufgerufen wird. Es wurde auch eine `validateTemperatureRequestBody()`-Middleware implementiert, um den Payload für Temperaturdaten zu validieren.

requireInternalApiKey

`requireInternalApiKey()` schützt Endpoints, die nur von internen Services, wie z.B. dem Flask-Server, aufgerufen werden dürfen und nicht von normalen Clients. Die Middleware überpürft somit, ob der API-Key im gesendeten Header vorhanden ist und vergleicht ihn mit dem Wert der in .env gespeicherten Umgebungsvariable `INTERNAL_API_KEY` (siehe 2.1.6). Bei einem erfolgreichen Vergleich wird die Anfrage weitergeleitet, ansonsten wird der Fehlercode 403 mit der Nachricht `Access denied: Invalid API key` zurückgesendet.

Middleware-Reihenfolge

1. `authenticateToken()`: Benutzer ist angemeldet
2. `authorizeRole()`: Benutzer hat erforderliche Rolle

3. `validateBody()`: Validierung des Request-Bodys
4. `requireInternalApiKey()`: wenn interner API-Key gebraucht wird

4.3.4 Benutzerverwaltung und Authentifizierung

Überblick

Die Benutzerverwaltung und Authentifizierung bildet die Grundlage des gesamten Backends und kontrolliert nicht nur die Anmeldung von Benutzern, sondern auch die Autorisierung für alle nachfolgenden Operationen. Das System besteht aus vier eng zusammenhängenden Komponenten. Der `user.router.ts` definiert alle API-Endpoints für Registrierung, Login und Userverwaltung. Das `user.repository.ts` verwaltet alle Datenbankoperationen zu Benutzer-Abfragen und -Änderungen. Der `auth.service.ts` behandelt sicherheitskritische Operationen wie Passwort-Hashing mit Bcrypt (siehe 2.5.2) und Passwort-Vergleiche. `jwt-config.ts` konfiguriert zentral die JWT-Signierung und -Verifizierung mit dem HS256-Algorithmus (siehe 2.4.5). Die `user_data-Tabelle` in MSSQL (siehe 2.4.5) speichert alle Benutzerdaten persistent und stellt die Grundlage für alle Authentifizierungs- und Autorisierungsvorgänge dar (siehe 2.1.5). Das System implementiert ein sicheres Design, bei dem Passwörter niemals im Klartext gespeichert werden und die Authentifizierung vollständig auf Tokens basiert.

Listing 86: Beispiel: Model des Users

```
interface User {
    id: number;
    username: string;
    password_hash: string;
    created_at: Date;
    role: string;
}
```

User Repository - Datenbankoperationen

Das user.repository.ts enthält alle Funktionen für die Datenbankinteraktion mit Benutzern.

getAllUsers(): User[] retourniert alle Benutzer des Systems

Listing 87: getAllUsers

```
export async function getAllUsers(): Promise<User[]> {
    try {
        const result = await sql.query`SELECT id, username, password_hash, created_at,
            role FROM user_data`;
        return result.recordset.map(r => ({
            id: r.id,
            username: r.username,
            password_hash: r.password_hash,
            created_at: r.created_at,
            role: r.role
        })) as User[];
    } catch (error) {
        console.error("Error fetching all users:", error);
        throw error;
    }
}
```

getAllUsers(id: number): User | undefined retourniert einen User anhand seiner ID bzw. `undefined`, sollte kein User mit der übergebenen ID gefunden werden

Listing 88: getUser

```
export async function getUser(id: number): Promise<User | undefined> {
    try {
        const result = await sql.query`SELECT id, username, password_hash, created_at,
            role FROM user_data WHERE id = ${id}`;
        const r = result.recordset[0];
        if (!r) return undefined;
        return {
            id: r.id,
            username: r.username,
            password_hash: r.password_hash,
            created_at: r.created_at,
            role: r.role
        };
    } catch (error) {
        console.error("Error fetching user by ID:", error);
        throw error;
    }
}
```

addUser(username: string, password_hash: string): User erstellt einen User mit dem übergebenen Benutzernamen und Passwort-Hash und retourniert das erstellte `'User'`-Objekt

Listing 89: addUser

```
export async function addUser(username: string, password_hash: string): Promise<User> {
    try {
        const role = 'viewer';
        const result = await sql.query`
            INSERT INTO user_data (username, password_hash, role)
            OUTPUT INSERTED.id, INSERTED.username, INSERTED.password_hash,
                INSERTED.created_at, INSERTED.role
            VALUES (${username}, ${password_hash}, ${role})
        `;
        const r = result.recordset[0];
        return {
            id: r.id,
            username: r.username,
            password_hash: r.password_hash,
            created_at: r.created_at,
            role: r.role
        };
```

```
17      } catch (error) {
18          console.error("Error creating a new user:", error);
19          throw error;
20      }
21  }
```

updateUser(user: UpdateUserInput): boolean updatet einen bereits existierenden User und gibt den Status der Aktion zurück. `true` bei erfolgreichem Ändern, `false` wenn ein Fehler aufgetreten ist. Damit Felder die nicht geupdatet werden nicht als `required` wahrgenommen werden, wurde ein `UpdateUserInput`-Interface angelegt, welches nur die zu updatenden Attribute enthält.

Listing 90: UpdateUserInput-Interface

```
1  export interface UpdateUserInput {
2      id: number;
3      username: string;
4      password_hash: string;
5  }
```

Listing 91: updateUser

```
1  export async function updateUser(user: UpdateUserInput): Promise<boolean> {
2      try {
3          const result = await sql.query`
4              UPDATE user_data
5              SET username = ${user.username}, password_hash = ${user.password_hash}
6              WHERE id = ${user.id}
7          `;
8          return result.rowsAffected[0] > 0;
9      } catch (error) {
10         console.error("Error updating user:", error);
11         throw error;
12     }
13 }
```

updateUserRole(id: number, newRole: string): boolean updatet die Rolle eines Benutzers. Wenn die Änderung erfolgreich war, wird `true` retourniert, ansonsten `false`. Bei einer Degradierung vom Admin zum Viewer wird der entsprechende Role Request auch auf `declined` gesetzt.

Listing 92: updateUserRole

```
1  export async function updateUserRole(id: number, newRole: string): Promise<boolean> {
2      try {
3          // Fetch the old role
4          const userResult = await sql.query`SELECT role FROM user_data WHERE id = ${id}`;
5          const oldRole = userResult.recordset[0]?.role;
6          const result = await sql.query`
7              UPDATE user_data
8              SET role = ${newRole}
9              WHERE id = ${id}
10         `;
11         // If changing from admin to viewer, decline the associated request
12         if (oldRole === 'admin' && newRole === 'viewer') {
13             await sql.query`
14                 UPDATE role_requests
15                 SET status = 'declined', updated_at = SYSDATETIMEOFFSET()
16                 WHERE user_id = ${id} AND status = 'accepted'
17             `;
```

```
18          }
19          return result.rowsAffected[0] > 0;
20      } catch (error) {
21          console.error("Error updating user role:", error);
22          throw error;
23      }
24  }
```

deleteUser(id: number): boolean löscht einen Benutzer und retourniert bei Erfolg `true`, ansonsten `false`

Listing 93: deleteUser

```
1  export async function deleteUser(id: number): Promise<boolean> {
2      try {
3          const result = await sql.query`DELETE FROM user_data WHERE id = ${id}`;
4          return result.rowsAffected[0] > 0;
5      } catch (error) {
6          console.error("Error deleting user:", error);
7          throw error;
8      }
9  }
```

Authentication Service - Passwort-Sicherheit

`auth.service.ts` ist verantwortlich für sicherheitskritische Operationen bezüglich Passwörter und Authentifizierung. Die Funktion `hashClientHash(clientHash)` nimmt den vom Client gesendeten Passwort-Hash entgegen und hasht ihn erneut mit Bcrypt. Die Anzahl der Bcrypt-Runden wird aus der Umgebungsvariable `BCRYPT_ROUNDS` geladen und standardmäßig auf 12 gesetzt. Dieser Wert bietet ein gutes Gleichgewicht zwischen Sicherheit und Performance. Der resultierende Bcrypt-Hash wird in der Datenbank gespeichert. Die Funktion `comparePassword(clientHash, storedBcryptHash)` vergleicht einen eingehenden Passwort-Hash mit dem gespeicherten Bcrypt-Hash mittels `bcrypt.compare()` und gibt `true` zurück, wenn sie übereinstimmen. Diese Funktion prüft zusätzlich, ob der gespeicherte Hash mit `$2` beginnt, was auf einen gültigen Bcrypt-Hash hindeutet, und lehnt ab, wenn dies nicht der Fall ist. Dies verhindert, dass ungültige Hashes verglichen werden.

Listing 94: auth.service.ts

```
1  // This file provides authentication-related helper functions.
2  // It includes password hashing and comparison logic for login and registration.
3
4  import bcrypt from 'bcrypt';
5
6  const ROUNDS = Number(process.env.BCRYPT_ROUNDS || 12);
7
8  /**
9   * Hashes the client-provided password hash (clientHash) using bcrypt for server-side
        storage.
10  *
11  * The application expects the client to send a clientHash (e.g. a client-side hash or raw
        password
12  * depending on your frontend). This function applies bcrypt with server-configured rounds
        and
13  * returns the resulting bcrypt hash suitable for persistent storage.
```

```
14   */
15  export async function hashClientHash(clientHash: string): Promise<string> {
16      return bcrypt.hash(clientHash, ROUNDS);
17  }
18
19  /**
20   * Compares the client-provided password hash (clientHash) against the stored bcrypt hash.
21   */
22  export async function comparePassword(clientHash: string, storedBcryptHash: string):
        Promise<boolean> {
23      // Only accept stored bcrypt hashes; compare using bcrypt.compare
24      if (!storedBcryptHash.startsWith('$2')) {
25          return false;
26      }
27
28      return bcrypt.compare(clientHash, storedBcryptHash);
29  }
```

JWT-Konfiguration

`jwt-config.ts` zentralisiert die gesamte JWT-Konfiguration der Applikation (siehe 2.4.5). Der `jwtAlgorithm` ist fest auf 'HS256' gesetzt, was einen symmetrischen Verschlüsselungsalgorithmus mit einem gemeinsamen Secret darstellt. Das Secret wird aus der Umgebungsvariable `JWT_SECRET` aus `.env` geladen und muss für die Applikation gesetzt sein. Wenn es nicht gesetzt ist, wird ein Fehler geworfen und die Applikation startet nicht. Der `jwtSignKey` und `jwtVerifyKey` sind beide auf das Secret aus der Umgebung gesetzt, da bei HS256 das Secret für Signing und Verifying identisch ist. Die Hilfsfunktionen `getJwtSignOptions()` und `getJwtVerifyOptions()` geben standardisierte Option-Objekte zurück, die den HS256-Algorithmus erzwingen. Nach dem Erstellen eines Access Tokens durch z.B. einen Login, wird es bei den Requests des Users mitgesendet, wodurch sich dieser authentifizieren und für gewisse Aktionen, welche beispielsweise uneingeloggten Benutzern nicht zustehen, autorisieren kann (siehe 4.3.3 und 4.3.5.

Listing 95: jwt-config.ts

```
1   import type { SignOptions, VerifyOptions } from 'jsonwebtoken';
2
3   // This module centralizes JWT configuration for the application.
4   // It intentionally enforces a symmetric HS256 setup using a shared secret
5   // (process.env.JWT_SECRET). The module preserves the previous public API
6   // shape (jwtSignKey, jwtVerifyKey, getJwtSignOptions, getJwtVerifyOptions)
7   // so callers in other files don't need to change.
8
9   const secretFromEnv = process.env.JWT_SECRET; // HS256 Secret (required)
10
11  // Force HS256 with a shared secret.
12  export const jwtAlgorithm: 'HS256' = 'HS256';
13
14  if (!secretFromEnv) {
15    throw new Error('[FATAL] JWT_SECRET is not set in environment. Please set
          process.env.JWT_SECRET');
16  }
17
18  // Sign/verify key: simple symmetric secret (HS256)
19  export const jwtSignKey: string = secretFromEnv as string;
20  export const jwtVerifyKey: string = secretFromEnv as string;
21
22  // Option helpers for sign/verify calls; keeps callers explicit about alg/opts
23  export function getJwtSignOptions(): SignOptions {
24    return { algorithm: jwtAlgorithm } as SignOptions;
25  }
26
27  export function getJwtVerifyOptions(): VerifyOptions {
```

```
28    return { algorithms: [jwtAlgorithm] } as VerifyOptions;
29  }
```

User Router - API Endpoints

GET /api/users retourniert alle User im System bis auf den `Superuser`. Diese Route kann durch `requireSuperuser` und `authenticateToken` (siehe 4.3.3) nur von einem angemeldeten Superuser aufgerufen werden.

GET /api/users/me retourniert die Daten des aktuellen Users. Die Route kann ebenfalls nur von einem eingeloggten User aufgerufen werden, was durch `authenticateToken` sichergestellt wird.

GET /api/users/:id retourniert die Daten des Users mit der übergebenen ID. `requireSuperuser` und `authenticateToken` legen fest, dass die Route nur vom eingeloggten Superuser aufgerufen werden kann.

POST /api/users registriert einen neuen User. Dabei werden die Daten aus dem Request Body entnommen und inklusive des gehashten Passworts in der Datenbank persistiert. Es wird auch ein Access Token aus den Benutzerdaten erstellt und ein Refresh Token mittels `crypto.randomBytes()` generiert (siehe 2.4.5 und 2.5.2). Die Tokens werden dann gemeinsam mit einem Ablaufdatum zu einer Session für den neuen Benutzer zusammengefasst, wobei ein Access Token zwei Stunden und ein Refresh Token einen Tag lang gültig ist, und in der Datenbank gespeichert.

Listing 96: Tokens und Session werden erstellt und persistiert

```
1         const accessToken = jwt.sign({ id: newUser.id }, signKey, { expiresIn: '2h',
              ...getJwtSignOptions() });
2         const refreshToken = crypto.randomBytes(64).toString('hex');
3         const expiresAt = new Date(Date.now() + 24 * 60 * 60 * 1000); // refresh token: 1
              day
4         await sessionRepo.saveSession(accessToken, newUser.id, expiresAt, refreshToken);
5         res.status(StatusCodes.CREATED).json({ accessToken, refreshToken });
```

POST /api/users/login loggt den Benutzer ein. Mittels `auth.service.ts` wird der Hashwert des eingegebenen Passworts mit dem gespeicherten verglichen. Wenn der Vergleich erfolgreich war, ist der User eingeloggt und bekommt wieder einen neuen Access- und Refresh Token. Sollte er bei mehr als drei Geräten gleichzeitig angemeldet sein, wird er von einem Gerät abgemeldet, sodass nur drei gleichzeitge Sessions möglich sind (siehe 4.3.5).

Listing 97: Passwortvergleich

```
1 const isValid = await authService.comparePassword(clientHash, user.password_hash);
2         if (!isValid) {
3             return res.status(StatusCodes.UNAUTHORIZED).json({ error: 'Invalid
                  credentials' });
4         }
```

POST /api/users/refresh erneuert die Lebenszeit eines Access Tokens bzw. gibt dem User einen neuen, solange der Refresh Token noch gültig ist.

DELETE /api/users/session loggt den aufrufenden Benutzer aus und löscht die dazugehörende Session aus der Datenbank.

PUT /api/users/:id lässt den Superuser die Benutzerdaten des Users mit der übergebenen ID bearbeiten. Somit können der Benutzername und das Passwort geändert werden.

DELETE /api/users/:id lässt den Superuser den Benutzer mit der übergebenen ID löschen. Dabei werden auch alle aktiven Sessions von diesem aus der Datenbank gelöscht.

PUT /api/users/:id/role lässt den Superuser die Rolle des Benutzer mit der übergebenen ID von `viewer` zu `admin` ändern und umgekehrt.

PATCH /api/users/me/alarm ändert für den aufrufenden eingeloggten User, ob dieser für eine spezifische Kamera über Alarme benachrichtigt wird.

4.3.5 Sessionverwaltung und Token-Persistierung

Überblick

Das Session-Management ist die Brücke zwischen JWT-Tokens und der Datenbank und implementiert die serverseitige Kontrolle über Token-Gültigkeit. Während ein JWT theoretisch stateless ist, also jeder Server könnte es verifizieren, ohne auf die Datenbank zuzugreifen, würde dies bedeuten, dass ein gehacktes Token bis zum Ablauf von diesem gültig bleibt. Das Session-System löst dieses Problem, indem es jeden Token-Eintrag in der `sessions`-Tabelle speichert und bei jeder Anfrage validiert. Dies ermöglicht es dem Server, Tokens z.B. beim Logout sofort zu widerrufen, abgelaufene Sessions zu identifizieren und zu löschen, sowie die Anzahl der gleichzeitigen Sessions pro Benutzer zu begrenzen. Das `session.repository.ts`

verwaltet alle Datenbankoperationen rund um Sessions und das `session.model.ts` definiert die Datenstruktur.

Listing 98: Session-Model

```
interface Session {
    id?: number;
    token: string;
    user_id: number;
    expires_at: Date;
    refresh_token: string;
}
```

Session-Repository

Das `session.repository.ts` enthält alle Funktionen für Session-Verwaltung in der Datenbank.

saveSession(token, userId, expiresAt, refreshToken): void speichert eine neue Session oder aktualisiert eine existierende. Die Funktion versucht zuerst, einen existierenden Session-Eintrag mit dem übergebenen Token zu finden und aktualisieren. Falls dies erfolgreich ist, wird die Funktion beendet. Anderenfalls wird ein neuer Eintrag eingefügt (siehe 96).

getSessionByToken(token): Session | undefined sucht eine Session nach ihrem accessToken und gibt das Session-Objekt zurück oder undefined. Sie ist die Basis-Funktion für die Authentifizierungs-Middleware.

getSessionByRefreshToken(refreshToken): Session | undefined sucht eine Session nach dem Refresh-Token und wird verwendet, wenn der Client ein neues AccessToken anfordert.

deleteSession(token): void löscht eine Session aus der Datenbank, was meistens durch einen Logout oder Cleanup-Job passiert.

deleteAllSessionsForUser(userId: number): void löscht alle Sessions eines bestimmten Users aus der Datenbank, was meistens beim Löschen eines Benutzers verwendet wird

updateAccessTokenByRefreshToken(refreshToken, newAccessToken): boolean aktualisiert `token` einer existierenden Session, behält aber den Refresh Token. Dies wird verwendet, wenn der Client das AccessToken erneuert, aber nicht erneut authentifizieren möchte.

trimSessionsForUser(userId, maxSessions): void limitiert die Anzahl aktiver Sessions pro Benutzer. Falls ein Benutzer mehr als drei Sessions hat, werden die ältesten Einträge gelöscht, um wenige gleichzeitige Sessions pro Benutzer zu erzwingen.

Listing 99: trimSessionsForUser

```
export async function trimSessionsForUser(userId: number, maxSessions: number):
    Promise<void> {
    try {
        const sessions = await getSessionsByUser(userId);
        if (sessions.length <= maxSessions) return;
        const excess = sessions.slice(maxSessions);
        for (const s of excess) {
            await deleteSession(s.token);
        }
    } catch (error) {
        console.error(`Error trimming sessions for user ${userId}:`, error);
    }
}
```

4.3.6 Verwaltung der Rollenanfragen

Das Role-Request-System implementiert einen strukturierten Genehmigungsworkflow für Rollenerhöhungen von `viewer` zu `admin`. Da nicht jeder Benutzer automatisch Admin-Rechte haben sollte, muss ein `viewer` eine Anfrage stellen, die ein `super_user` genehmigen oder ablehnen kann. Dies verhindert unbefugten Zugriff auf kritische bzw. privilegierte Funktionen und sorgt für Kontrollierbarkeit in diesem Multi-User-Systemen (siehe 7).

Listing 100: role-request.model.ts

```
export type RoleRequestStatus = 'pending' | 'accepted' | 'declined';

export interface RoleRequest {
    id: number;
    user_id: number;
    requested_role: string;
    status: RoleRequestStatus;
    created_at: Date;
    updated_at: Date;
}
```

Role Request Repository

getPendingRoleRequestsWithUsernames(): any[] retourniert alle offenen Rollenänderungsanträge inklusive den Namen des Users, der die Role Request gestellt hat [101]

Listing 101: getPendingRoleRequestsWithUsernames

```
    const result = await sql.query`
            SELECT rr.*, u.username
            FROM role_requests rr
            JOIN user_data u ON rr.user_id = u.id
            WHERE rr.status = 'pending'
    `;
    return result.recordset;
```

updateRoleRequestStatus(id: number, status: RoleRequestStatus): RoleRequest updatet den Status einer Role Request anhand der ID [102]

Listing 102: updateRoleRequestStatus

```
UPDATE role_requests
SET status = ${status}, updated_at = SYSDATETIMEOFFSET()
OUTPUT INSERTED.*
WHERE id = ${id}
```

getRoleRequestById(id: number): RoleRequest retourniert den Role Request mit der übergebenen ID

createRoleRequestIfNone(user_id: number, requested_role: string): RoleRequest : erstellt einen Role Request für den User mit der übergebenen ID bzw. updatet seinen Antrag, wenn für ihn bereits einer in der Datenbank gespeichert ist (z.B. von `'declined'` auf `'pending'`) [103]

Listing 103: createRoleRequestIfNone

```
        // Strict: exactly one request per user max
        const existingResult = await sql.query`SELECT * FROM role_requests WHERE user_id =
            ${user_id}`;
        const existing = existingResult.recordset[0];
        if (existing) {
            // Update existing request: set new requested_role, reset status to pending
                and update timestamp
            const updateResult = await sql.query`
                UPDATE role_requests
                SET requested_role = ${requested_role}, status = 'pending', updated_at =
                    SYSDATETIMEOFFSET()
                OUTPUT INSERTED.*
                WHERE user_id = ${user_id}
            `;
            return updateResult.recordset[0] || null;
        }
        const insertResult = await sql.query`
            INSERT INTO role_requests (user_id, requested_role, status, created_at,
                updated_at)
            OUTPUT INSERTED.*
            VALUES (${user_id}, ${requested_role}, 'pending', SYSDATETIMEOFFSET(),
                SYSDATETIMEOFFSET())
        `;
        return insertResult.recordset[0] || null;
```

getRoleRequestByUserId(user_id: number): RoleRequest | undefined retourniert den Role Request zu dem User mit der übergebenen ID

Role Request Router

POST / erstellt für den aufrufenden User einen Role Request Antrag bzw. updatet seinen, wenn schon einer vorhanden ist

GET /pending retourniert alle Role Requests mit dem Status `'pending'`, steht nur dem Super User zur Verfügung

PATCH /:id akzeptiert oder lehnt einen spezifischen Role Request ab und aktualisiert die Rolle des Users dementsprechend, ebenfalls `requireSuperuser` [4.3.3]

GET /my-request retourniert den Role Request des aufrufenden Users

4.3.7 Kameraverwaltung

Das Kameraverwaltungs-System ist eine zentrale Komponente des Backends und implementiert die Verwaltung von physischen Kameras sowie die Zugriffskontrolle darauf. Während die `cameras`-Tabelle die eigentlichen Kamera-Informationen speichert, definiert die `user_cameras`-Tabelle, welche Benutzer auf welche Kameras zugreifen dürfen (siehe 2.1.5). Dies ermöglicht ein Berechtigungssystem, bei dem jeder Benutzer nur die für ihn relevanten Kameras sieht und steuert. Das System speichert auch sensitive Kamera-Zugangsdaten verschlüsselt mit RSA (siehe 2.1.6) und bietet Endpoints für Adminis zum Erstellen/Löschen von Kameras sowie für Benutzer zum Verknüpfen und Konfigurieren ihrer Kameras.

Listing 104: Camera-Model

```
1      export interface Camera {
2      // Unique identifier of the camera record in the database
3      id: number;
4      // Human-readable display name shown in the app (unique)
5      camera_name: string;
6      // Stream endpoint to access the camera feed (e.g., RTSP/HTTP URL)
7      stream_url: string;
8      // Device login username used to authenticate against the camera
9      camera_username: string;
10     // Camera password encrypted on the client (RSA-OAEP, base64). Stored
           encrypted-at-rest;
11     // only decrypted on the backend when needed (e.g., for Flask registration).
12     camera_pw_encrypted: string;
13 }
```

Listing 105: User_Camera-Model

```
1  export interface UserCamera {
2      user_id: number;
3      camera_id: number;
4      camera_lens: string;
5      is_alarm_active: boolean;
6  }
```

Camera Repository und Router

getCamerasByUser(user_id: number): UserCamera[] retourniert alle verlinkten Kameras eines spezifischen Benutzers, Router: `GET /`

getAllCameras(): Camera[] retourniert alle gespeicherten Kameras im System ohne deren Zugangsdaten, dafür sorgt `Promise<Omit<Camera, 'camera_username' | 'camera_pw_encrypted'>`, funktioniert ohne sich authentifizieren zu müssen, Router: `GET /all`

getAllCamerasWithCredentials(): Camera[] retourniert alle verlinkten Kameras im System inklusive Zugangsdaten, diese müssen jedoch noch entschlüsselt werden (siehe 2.1.6). Außerdem braucht man für diese Operation den internen API-Key aus .env, Router: `GET /all-confidential, requireInternalApiKey` (siehe 4.3.3)

getCameraByNameAndUrl(name: string, url: string): Camera retourniert die Kamera mit dem übergebenen Namen und URL ohne den Zugangsdaten von dieser

getCameraById(id: number): Camera retourniert die Kamera mit der übergebenen ID

linkUserToCamera(user_id: number, camera_id: number, camera_lens: string): boolean verknüpft einen User mit einer Kamera aus dem System und speichert auch eine beliebige Ansichtsart [2.1.5], Router: `POST /link` verlinkt den aktuellen User mit der ausgewählten Kamera

decryptCameraPassword(encryptedBase64: string): string entschlüsselt das übergebene Kamerapasswort mittels private.pem (siehe 2.1.6 und 2.5.2)

Listing 106: Entschlüsselung des Kamerapassworts

```
export function decryptCameraPassword(encryptedBase64: string): string {
    const privateKey = fs.readFileSync('./private.pem', 'utf8');
    const buffer = Buffer.from(encryptedBase64, 'base64');
    const decrypted = crypto.privateDecrypt(
        {
            key: privateKey,
            padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
        },
        buffer
    );
    return decrypted.toString('utf8');
}
```

createCameraIfNotExists(camera: Camera): camera: Camera, alreadyExists: boolean erstellt eine Kamera, insofern sie noch nicht existiert, und registriert sie beim Flask Server mittels API-Key, damit dieser die Temperaturdaten der Kamera empfangen und verabeiten kann. `alreadyExists` ist eine zusätzliche Flag, die indiziert, ob die erstellte Kamera bereits existiert

hat. Router: `POST /create`, `validateBody`, `authorizeRole('admin', 'super_user')` macht den Endpunkt nur für Admins und den Super User zugänglich (siehe 4.3.3)

deleteUserCamera(user_id: number, camera_id: number): boolean löscht die Verknüpfung zwischen einem User und einer Kamera, Router: `DELETE /:id, authenticateToken` [4.3.3]

deleteAllCamerasForUser(user_id: number): number löscht alle Verknüpfungen eines Benutzers zu den Kameras und retourniert die Anzahl der gelöschten Links, wird z.B. verwendet wenn ein Benutzer gelöscht wird, Route `DELETE /`, `authenticateToken` [4.3.3]

updateCamera(camera: Camera): boolean updatet die übergebene Kamera anhand ihrer ID, Router: `PATCH /:id`, `authenticateToken, authorizeRole('admin, super_user'`, validateBody) [4.3.3]

deleteCameraCompletely(camera_id: number): boolean löscht eine Kamera anhand ihrer ID komplett aus dem System, inklusive Verknüpfungen zu Usern, Temperaturdaten und der Registrierung beim Flask Server. Route: `DELETE /full/:id`, `authenticateToken, authorizeRole('admin, super_user')` [4.3.3]

isUserLinkedToCamera(user_id: number, camera_id: number): boolean überprüft, ob ein User mit einer Kamera verknüpft ist und retourniert das erhaltene Ergebnis als `boolean`-Wert, Router: `POST /link`, `authenticateToken` [4.3.3]

updateCameraName(camera_id: number, new_name: string): boolean ändert anhand der ID der Kamera ihren Namen, Router: `PATCH /:id/change-name`, `authenticateToken, authorizeRole('admin, super_user')` [4.3.3]

getUserCamera(user_id: number, camera_id: number): UserCamera | undefined retourniert anhand der übergebenen `user_id` und `camera_id` den dazu passenden `user_cameras`-Eintrag, insofern dieser existiert. Wird keiner gefunden, wird `undefined` retourniert.

updateUserCameraAlarmFlag(user_id: number, camera_id: number, isActive: boolean): boolean ändert für eine bestimmte `user_cameras`-Verknüpfung, ob dieser Benutzer bei einem Alarm dieser Kamera benachrichtigt wird, Router [4.3.4]

updateUserCameraLens(user_id: number, camera_id: number, camera_lens: string): boolean ändert für eine bestimmte `user_cameras`-Verknüpfung, welche Ansicht der Kamera der Benutzer angezeigt bekommt (`'normal', 'thermal'`), Router: `PATCH /:id/lens`, `authenticateToken` [4.3.3]

4.3.8 Temperaturdatenverwaltung

Das Temperature-Data-System ist spezialisiert auf die Speicherung und Abfrage großer Mengen von Zeitreihendaten, die von Kameras kontinuierlich gemessen werden.

Listing 107: TemperatureData-Model

```
export interface TemperatureData {
    time: Date; // timestamp when the measurement was taken
    temperature: number; // temperature value in C
    camera_id: number; // foreign key reference to camera
    confidence: number; // AI confidence between 0 and 1
    is_risky: boolean; // fire risk detected
}
```

TemperatureData-Repository und -Router

insertTemperatureData(data: TemperatureData): void persistiert einen neuen Temperaturwert in der Datenbank, Router: `POST /, requireInternalApiKey, validateTemperatureRequestBody` [4.3.3]

getAllTemperatureData(): TemperatureData[] retourniert alle gespeicherten Temperaturwerte, absteigend nach der Zeit sortiert, Router: `GET /`

deleteTemperatureDataByCameraId(camera_id: number): number löscht alle Temperaturdaten einer gewissen gewissen Kamera und retourniert die Anzahl an gelöschten Einträgen, wird beim Löschen einer Kamera verwendet

deleteOldTemperatureData(days: number = 7): number löscht alle Temperaturdaten, die älter als eine gewisse Tagesanzahl sind. Die Funktion kommt bei Datenbereinigungen wie Cron-Jobs zum Einsatz (siehe 2.5.2 und 4.3.1)

getTemperatureDataBetween(start: Date, end: Date, camera_id: number): TemperatureData[] retourniert alle Temperaturdaten einer gewissen Kamera, die zwischen zwei verschiedenen Zeitpunkten aufgezeichnet wurden, , Router: `GET /between`

getNewestTemperatureData(camera_id: number): TemperatureData | null gibt den neuesten Temperaturwert einer spezifischen Kamera zurück, Router: `GET /newest-temp-data`

GET /day retourniert alle Temperaturdaten, die an den übergebenen Tag verzeichnet wurden

POST /flask empfängt die Temperaturdaten der Kameras die der Flask-Server ans Backend sendet und persistiert diese

5 Aufgabenverteilung

Hier wird illustriert, welches Teammitglied für welchen Teil der Arbeit verantwortlich ist. Diese Zuteilung betrifft die Verfassung sowie den Praxisteil.

5.1 Samuel Riss-Stelzmüllner

Inhaltsverzeichnis Riss

1 Einleitung **1**

1.1 Ausgangslage 1

1.2 Zielsetzung 1

1.3 Projektumfeld 2

1.3.1 Projektteam 2

1.3.2 Betreuung 2

1.3.3 Auftraggeber 3

2 Grundlagen und Methoden **4**

2.2 Fachbegriffe 27

2.2.1 Data Augmentation 27

2.2.2 Thread 27

2.2.3 Neuronales Netzwerk 27

2.2.4 Klassifikation 29

2.2.5 Features 30

2.2.6 Supervised Learning 30

2.2.7 LSTM 31

2.3 Verwendete Entwicklungssysteme 32
2.4 Verwendete Technologien 36
2.4.8 branderkennungskomponente 46
2.5 verwendete bibliotheken und plug-Ins 48
2.5.1 Branderkennungskomponente 48
2.6 verworfene Optionen 55
2.6.1 Trainingsdaten durch Data Augmentation 55

3 Ergebnis **56**

3.4 Branderkennungskomponente 58

4 Implementierung **59**

4.1 Branderkennungskomponente 59
4.1.1 Testumgebung 59
4.1.2 Datenerfassung 60
4.1.3 Visualisierung 61
4.1.4 Trainingsdaten 62
4.1.5 Modelltraining 64
4.1.6 Flask Server 65

5.2 Simon Brandstaetter

Inhaltsverzeichnis Brandstätter

1 Einleitung **1**

2 Grundlagen und Methoden **4**

2.1 Architektur 4
2.1.1 Frontend 4

2.3 Verwendete Entwicklungssysteme 32
2.3.2 Webstorm 33
2.3.3 Visual Studio Code 34
2.4 Verwendete Technologien 36
2.4.7 frontend 43
2.5 verwendete bibliotheken und plug-Ins 48
2.5.3 Frontend 53

3 Ergebnis **56**
3.1 Architektur 56
3.2 Frontend 57

4 Implementierung **59**
4.2 Frontend 71
4.2.1 Komponentenaufbau 71
4.2.2 Routing und Navigation 71
4.2.3 API-Anbindungen und Services 74
4.2.4 Formularlogik und Validierung 78
4.2.5 Funktionen 78
4.2.6 Fehlerbehandlung 85
4.2.7 Kamera-Feed Proxy 86

5.3 Eric Baumgartner

Inhaltsverzeichnis Baumgartner

1 Einleitung **1**

2 Grundlagen und Methoden **4**

2.1 Architektur 4

2.1.2 Backend 11

2.1.3 Projektstruktur des Backends 11

2.1.4 Rollenmodell 16

2.1.5 Aufbau der Datenbanken 18

2.1.6 Umgebungskonfiguration und Sicherheit 26

2.3 Verwendete Entwicklungssysteme 32

2.3.1 Git 32

2.3.4 Docker Desktop 34

2.3.5 WSL 35

2.4 Verwendete Technologien 36

2.4.1 Mobotix MX-M16TB-R079 36

2.4.2 Backend 36

2.4.3 npm 37

2.4.4 OpenAPI 39

2.4.5 JWTs 39

2.4.6 TimescaleDB 42

2.5 verwendete bibliotheken und plug-Ins 48

2.5.2 Backend 50

3 Ergebnis **56**

3.3 Backend 58

4 Implementierung **59**

4.3 Backend 88

4.3.1 app.ts - Application Entry Point 88

4.3.2 OpenAPI Dokumentation 92
4.3.3 Middleware 93
4.3.4 Benutzerverwaltung und Authentifizierung 96
4.3.5 Sessionverwaltung und Token-Persistierung 102
4.3.6 Verwaltung der Rollenanfragen 104
4.3.7 Kameraverwaltung 106
4.3.8 Temperaturdatenverwaltung 109

6 Resümee

Durch das Erstellen dieser Diplomarbeit durften wir viele wertvolle Dinge lernen.
Wir lernten nicht nur, wie man eine wissenschaftliche Arbeit verfasst, sondern auch viele wichtige Lektionen über die Herangehensweise an ein Software-Projekt mit mehreren Teammitgliedern. Beispielsweise wäre uns viel Arbeit am Backend erspart geblieben, wenn wir eine ausführlichere Schnittstellendefinition zu Projektstart angefertigt hätten. Wir haben auch gelernt, mündliche Gespräche zu wichtigen Themen und Entscheidungen schriftlich festzuhalten und allgemein eine schlanke Dokumentation für uns selbst zu erstellen. Denn diese fehlende Dokumentation ist uns nach der Pause durch die Sommerferien gehörig in den Rücken gefallen.
Neben den Lektionen zur Projektarbeit lernten wir auch so einiges über die Abläufe in einer Firma mit den Dimensionen der Voest. Herr Buchegger ließ uns regelmäßig an Meetings teilnehmen, was uns viele interessante Einblicke in das Innenleben des Konzerns gab.

Glossar

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

API Application Programming Interface

bzw. beziehungsweise

CORS Cross-Origin Resource Sharing

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

CSV Comma-Seperated Values

DB Database

DBMS Database Management System

GNU GNU's Not Unix

HMAC Hash-based Message Authentication Code

HTML Hypertext Markup Language

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

JWT JSON Web Token

KI Künstliche Intelligenz

LSTM Long Short-Term Memory

ML Machine Learning

ms Millisekunden

ORDBMS Object-Relational Database Management System

ORM Object Relational Mapping

ReLU Rectified Linear Unit

REST Representational State Transfer

REST-API Representational State Transfer Application Programming Interface

RNN Recurrent Neural Networks

RxJS Reactive Extensions for JavaScript

SASS Syntactically Awesome Style Sheets

SCSS Sassy Cascading Style Sheets

SIMD Single Instruction, Multiple Data

SPA Single Page Applikation

SQL Structured Query Language

SVG Scalable Vector Graphic

TTF TrueType Font

u.a. unter anderem

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

usw. und so weiter

UTF-8 8-Bit Universal Coded Character Transformation Format

uvm. und vieles mehr

WSL Windows Subsystem for Linux

YAML YAML Ain't Markup Language

z.B. zum Beispiel

Literaturverzeichnis

[1] Stadt Linz, „Stadt Linz: Feuerwehren in Linz – Betriebsfeuerwehr voestalpine Standortservice GmbH," 2026, letzter Zugriff am 08.03.2026. Online verfügbar: https://www.linz.at/feuerwehr/3030.php

[2] ——, „Stadt Linz: Jahresbericht 2024 der Linzer Feuerwehren," 2025, letzter Zugriff am 08.03.2026. Online verfügbar: https://www.linz.at/medienservice/2025/files/PK20250306_Jahresbericht.pdf

[3] A. Schaffrinna, „Neues Logo Für Voestalpine," Letzter Zugriff am 20.03.2026. Online verfügbar: https://www.designtagebuch.de/neues-logo-fuer-voestalpine

[4] B. Lenaerts-Bergmans, „What is Privilege Escalation?" 2022, Letzter Zugriff am 21.03.2026. Online verfügbar: https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/privilege-escalation/

[5] A. Villavicencio, „Was ist das Least-Privilege-Prinzip?" 2022, Letzter Zugriff am 21.03.2026. Online verfügbar: https://www.crowdstrike.com/de-de/cybersecurity-101/identity-protection/principle-of-least-privilege-polp/

[6] Microsoft, „Data types (Transact-SQL)," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://learn.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql

[7] ——, „Indexes," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/indexes?view=sql-server-ver17

[8] ——, „Primary and foreign key constraints," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://learn.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-ver17

[9] ——, „Create check constraints," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://learn.microsoft.com/en-us/sql/relational-databases/tables/create-check-constraints?view=sql-server-ver17

[10] ——, „Unique constraints and check constraints," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://learn.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints?view=sql-server-ver17

[11] P. G. D. Group, „Indexes," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://www.postgresql.org/docs/current/indexes.html

[12] ——, „Data Types," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://www.postgresql.org/docs/current/datatype.html

[13] ——, „Constraints," 2025, Letzter Zugriff am 23.03.2026. Online verfügbar: https://www.postgresql.org/docs/current/ddl-constraints.html

[14] A. Mumuni und F. Mumuni, „Data augmentation: A comprehensive survey of modern approaches," *Array*, Vol. 16, S. 100258, 2022. Online verfügbar: https://www.sciencedirect.com/science/article/pii/S2590005622000911

[15] Google. (2023) Machine Learning Crash Course. Zugriff am 23.03.2026. Online verfügbar: https://developers.google.com/machine-learning/crash-course

[16] PyTorch Contributors, „BCEWithLogitsLoss," 2026, Zugriff am 24.03.2026. Online verfügbar: https://docs.pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html

[17] C. Olah, „Understanding LSTM Networks," 2015. Online verfügbar: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[18] GitLab, „What is Git version control?" 2025, Letzter Zugriff am 15.03.2026. Online verfügbar: https://about.gitlab.com/topics/version-control/what-is-git-version-control/

[19] J. Long, „Logo for Git," 2012, Letzter Zugriff am 15.03.2026. Online verfügbar: http://git-scm.com/downloads/logos

[20] Wikipedia, „Webstorm," 2024. Online verfügbar: https://de.wikipedia.org/wiki/WebStorm

[21] ——, „WebStorm Icon," Letzter Zugriff am 01.03.2026. Online verfügbar: https://de.wikipedia.org/wiki/Datei:WebStorm_Icon.png

[22] Wikimedia, „Visual Studio Code Logo," Letzter Zugriff am 18.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Visual_Studio_Code_1.35_icon.svg

[23] D. Inc., „Docker Desktop," 2025, Letzter Zugriff am 14.03.2026. Online verfügbar: https://docs.docker.com/desktop/

[24] ——, „Docker Logo," 2025, Letzter Zugriff am 14.03.2026. Online verfügbar: https://www.docker.com/company/newsroom/media-resources/

[25] M. Learn, „Was ist das Windows-Subsystem für Linux?" 2025, Letzter Zugriff am 15.03.2026. Online verfügbar: https://learn.microsoft.com/de-de/windows/wsl/about

[26] M. Corporation, „Logo for WSL," 2021, Letzter Zugriff am 15.03.2026. Online verfügbar: https://apps.microsoft.com/detail/9P9TQF7MRM4R?hl=en-us&gl=US

[27] MOBOTIX. (2023, Nov.) Technical Specifications MOBOTIX M16B Thermal-TR. Version V1.09_11/20/2023. Online verfügbar: https://www.mobotix.com/sites/default/files/2023-11/Mx_TS_M16TB-R_V1.09_EN.pdf

[28] C. Flavio, B. M. Potch *et al.*, „Introduction to Node.js," 2023, Letzter Zugriff am 26.01.2026. Online verfügbar: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

[29] N. C. B. Muenzenmeyer Brian, Crowdin Bot, „Node.js Logo," 2024, Letzter Zugriff am 27.01.2026. Online verfügbar: https://nodejs.org/en/about/branding/

[30] L. Karrys, M. B. Michael Rienstra, und E. Thomson, „About npm," 2023, Letzter Zugriff am 15.03.2026. Online verfügbar: https://docs.npmjs.com/about-npm

[31] Boboss74, „npm Logo," 2014, Letzter Zugriff am 15.03.2026. Online verfügbar: https://de.wikipedia.org/wiki/Datei:Npm-logo.svg

[32] R. Schimka, P. S. Alan Buchanan *et al.*, „Express/Node introduction," 2025, Letzter Zugriff am 12.03.2026. Online verfügbar: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/Introduction

[33] expressjs developers, „Express.js Logo," 2014, Letzter Zugriff am 12.03.2026. Online verfügbar: https://i.cloudup.com/zfY6lL7eFa-3000x3000.png

[34] I. Think-Redaktion, „What is REST API?" 2025, Letzter Zugriff am 13.03.2026. Online verfügbar: https://www.ibm.com/think/topics/rest-apis

[35] O. Initiative, „What is OpenAPI?" 2025, Letzter Zugriff am 16.03.2026. Online verfügbar: https://www.openapis.org/what-is-openapi

[36] ——, „Logo for OpenAPI," 2025, Letzter Zugriff am 16.03.2026. Online verfügbar: https://www.openapis.org/wp-content/uploads/sites/31/2018/02/OpenAPI_Logo_Pantone-1.png

[37] M. B. Jones und N. S. John Bradley, „JSON Web Token (JWT)," 2015, Letzter Zugriff am 12.03.2026. Online verfügbar: https://datatracker.ietf.org/doc/html/rfc7519

[38] K. Moriarty, J. J. Burt Kalinski, und A. Rusch, „SA Cryptography Specifications Version 2.2," 2016, Letzter Zugriff am 13.03.2026. Online verfügbar: https://datatracker.ietf.org/doc/html/rfc8017

[39] R. West, W. A. Mike Ray *et al.*, „What is SQL Server?" 2023, Letzter Zugriff am 22.02.2026. Online verfügbar: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver17

[40] Unbekannt, „Microsoft SQL Server Logo," 2010, Letzter Zugriff am 22.02.2026. Online verfügbar: http://www.volbase.co.uk/images%5Csqlserver%5Csql_server_2008_logo.png

[41] T. P. G. D. Team, „About PostgreSQL," 2026, Letzter Zugriff am 22.02.2026. Online verfügbar: https://www.postgresql.org/about/

[42] D. Lundin, „PostgreSQL Logo," 2008, Letzter Zugriff am 22.02.2026. Online verfügbar: http://pgfoundry.org/docman/view.php/1000089/124/elephant.svgz

[43] T. Data, „TimescaleDB Logo," 2026, Letzter Zugriff am 12.03.2026. Online verfügbar: https://assets.tigerdata.com/timescale-web/brand/tiger-data/flat-logos/logo-horizontal-black.svg

[44] D. Medchal, „Typescript vs JavaScript," Letzter Zugriff am 15.02.2026. Online verfügbar: https://www.openxcell.com/blog/typescript-vs-javascript/

[45] M. Sarbak, „HTML Logo," 2024. Online verfügbar: https://www.qarbon.it/blog/what-is-html

[46] Wikipedia, „HTML Logo," Letzter Zugriff am 28.02.2026. Online verfügbar: https://en.wikipedia.org/wiki/HTML

[47] J. Raimung, „SCSS zu CSS," Letzter Zugriff am 28.02.2026. Online verfügbar: https://kulturbanause.de/blog/einstieg-in-sass-funktionsweise-und-ueberblick

[48] Wikipedia, „TypeScript," 2026. Online verfügbar: https://de.wikipedia.org/wiki/TypeScript

[49] Pallets Projects, „Flask Documentation," 2026, Version 3.1.x, zuletzt abgerufen am 22.03.2026. Online verfügbar: https://flask.palletsprojects.com/

[50] Django Software Foundation, „Django Documentation," 2026, Version 6.0, zuletzt abgerufen am 22.03.2026. Online verfügbar: https://docs.djangoproject.com/en/6.0/

[51] Python Software Foundation, „threading — Thread-based parallelism," 2025, Zugriff am 21.03.2026. Online verfügbar: https://docs.python.org/3/library/threading.html

[52] ——, „Global Interpreter Lock (GIL)," 2025, Zugriff am 21.03.2026. Online verfügbar: https://docs.python.org/3/faq/library.html#what-is-the-global-interpreter-lock-gil

[53] ——. (2026) What is Python? Executive Summary. Online verfügbar: https://www.python.org/doc/essays/blurb/

[54] Bokeh Contributors, „Bokeh Documentation," https://docs.bokeh.org/en/latest/, 2026, Zugriff am 13.03.2026.

[55] W. McKinney, „Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt und J. Millman, Hrsgg., 2010, S. 56–61.

[56] NumPy Developers, „What is NumPy?" https://numpy.org/doc/stable/user/whatisnumpy.html, 2025, Zugriff am 15.03.2026.

[57] PyTorch Team, „PyTorch Documentation," 2025, Zugriff am 25.03.2026. Online verfügbar: https://docs.pytorch.org/

[58] A. Contributors, „Getting Started," 2026, Letzter Zugriff am 18.03.2026. Online verfügbar: https://axios-http.com/docs/intro

[59] P. Simek, W. M. Arthur Schreiber, und D. Hensby, „mssql," 2025, Letzter Zugriff am 21.03.2026. Online verfügbar: https://www.npmjs.com/package/mssql

[60] B. C, C. Rijk van Zanten *et al.*, „node-postgres," 2026, Letzter Zugriff am 21.03.2026. Online verfügbar: https://node-postgres.com/

[61] ——, „node-postgres," 2026, Letzter Zugriff am 21.03.2026. Online verfügbar: https://node-postgres.com/apis

[62] ——, „node-postgres," 2026, Letzter Zugriff am 21.03.2026. Online verfügbar: https://node-postgres.com/features

[63] ——, „node-postgres," 2026, Letzter Zugriff am 21.03.2026. Online verfügbar: https://node-postgres.com/guides

[64] T. Goode, „cors," 2026, Letzter Zugriff am 21.03.2026. Online verfügbar: https://expressjs.com/en/resources/middleware/cors.html

[65] J. Lowe, m. Scott Motte, und motdotenv, „dotenv," 2026, Letzter Zugriff am 21.03.2026. Online verfügbar: https://www.npmjs.com/package/dotenv

[66] Abhinav, „Understanding HTTP Status Codes with Express and Morgan," 2024, Letzter Zugriff am 21.03.2026. Online verfügbar: https://dev.to/abhivyaktii/understanding-http-status-codes-with-express-and-morgan-o71

[67] D. Arias, „Hashing in Action: Understanding bcrypt," 2021, Letzter Zugriff am 21.03.2026. Online verfügbar: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

[68] O. Foundation, „Node.js v25.8.1 documentation," 2025, Letzter Zugriff am 21.03.2026. Online verfügbar: https://nodejs.org/api/crypto.html

[69] Uptimia, „Planen Sie Cron-Jobs in Node.js mit Node-Cron," 2024, Letzter Zugriff am 21.03.2026. Online verfügbar: https://www.uptimia.com/de/learn/zeitplan-cron-jobs-in-node-js

[70] D. Clark und F. Kilcommins, „What Is OpenAPI?" 2025, Letzter Zugriff am 22.03.2026. Online verfügbar: https://swagger.io/docs/specification/v3_0/about/

[71] N. Dhandala, „How to Create Swagger Documentation for Node.js APIs," 2026, Letzter Zugriff am 22.03.2026. Online verfügbar: https://oneuptime.com/blog/post/2026-01-25-swagger-documentation-nodejs-apis/view

[72] R. Team, „RxJS Logo," Letzter Zugriff am 08.03.2026. Online verfügbar: https://rxjs.dev/

[73] E. Yalman, „CryptoJS Logo," Letzter Zugriff am 10.03.2026. Online verfügbar: https://medium.com/@emine.yalman/vue-js-ile-g%C3%BCvenli-%C5%9Fifreleme-cryptojs-ve-axios-kullan%C4%B1m%C4%B1-71d157fce925

[74] N. F. Team, „Node Forge Logo," Letzter Zugriff am 17.03.2026. Online verfügbar: https://nodeforge.io/

[75] Wikipedia, „Chart.js," 2025. Online verfügbar: https://en.wikipedia.org/wiki/Chart.js

[76] D. Balaji, „Chart.js Logo," Letzter Zugriff am 27.02.2026. Online verfügbar: https://dvmhn07.medium.com/introducing-chart-js-a-powerful-javascript-library-for-interactive-data-visualization-88add523d008

[77] M. AG, „mobotix help page," Letzter Zugriff am 5.3.2026. Online verfügbar: https://developer.mobotix.com/paks/help_cgi-image.html

[78] International Telecommunication Union, „Information technology – Digital compression and coding of continuous-tone still images (JPEG) – Requirements and guidelines," ITU-T Recommendation T.81, 1992, letzter Zugriff am 12.03.2026. Online verfügbar: https://www.w3.org/Graphics/JPEG/itu-t81.pdf

[79] Schaeffler. (2026) Bearing data. Schaeffler. Accessed 2026-03-14. Online verfügbar: https://medias-at.schaeffler.com/en/knowledge-center/rolling-bearings/bearing-data

[80] Pallets Projects, „Flask Documentation – Application Object: before_request," 2024, Abgerufen am 15. März 2026. Online verfügbar: https://flask.palletsprojects.com/en/stable/api/#flask.Flask.before_request

[81] PyTorch Contributors, „PyTorch Documentation," https://pytorch.org/docs/stable/, 2026, Accessed: 2026-03-23.

[82] Python Software Foundation, „The import system — Python 3 Language Reference," https://docs.python.org/3/reference/import.html, 2026, Accessed: 2026-03-23.

[83] R. Sharp, „nodemon," 2026, Letzter Zugriff am 22.03.2026. Online verfügbar: https://www.npmjs.com/package/nodemon

Abbildungsverzeichnis

1 Eric Baumgartner, Simon Brandstätter, Samuel Riss-Stelzmüllner (von links nach rechts) . . . 2
2 Voestalpine Logo . . . 3
3 Frontend Projektstruktur . . . 4
4 Frontend Models . . . 6
5 Figma Design . . . 8
6 Landing Page . . . 9
7 Kamera Liste . . . 9
8 Viewer Ansicht . . . 10
9 Add Camera . . . 10
10 Kameraelement Operationen als Admin . . . 11
11 Super User Manage Users . . . 11
12 Anzahl der Kamera Elemente . . . 11
13 Projektstruktur des Backends . . . 12
14 Ablauf eines User-Logins mit Abbildung der Kommunikation aller Komponenten 16
15 ERD des gesamten Datenmodells . . . 25
16 Logo von Git [19] . . . 33
17 Logo von Webstorm . . . 34
18 Logo von Visual Studio Code . . . 34
19 Logo von Docker Desktop [24] . . . 35
20 Logo von WSL [26] . . . 35
21 Logo von Node.js [29] . . . 36
22 Logo von npm [31] . . . 37
23 Logo von Express.js [33] . . . 38
24 Logo von OpenAPI [36] . . . 39
25 Logo von Microsoft SQL Server [40] . . . 41
26 Logo von PostgreSQL [42] . . . 42
27 Logo von TimescaleDB [43] . . . 43
28 Logo von Angular . . . 43
29 Logo von HTML [46] . . . 44
30 Kompilierung mittels SASS . . . 44
31 Typescript . . . 45
32 Logo von RxJS . . . 53
33 CryptoJS Logo . . . 53
34 Node Forge Logo . . . 54
35 Logo von Chart.js . . . 54
36 Logo von NGX-Scrollbar . . . 55
37 Architektur des MoboView Systems . . . 56
38 Blech wird erhitzt . . . 59
39 Setup Menü -> Thermalsensor Einstellungen . . . 60
40 Thermal Rohdaten aktiviert . . . 60
41 Temperaturverlauf des erhitzten Blechs . . . 62
42 sichere Sequenz mit kleiner Erhebung . . . 63

43 riskante Sequenz mit exponentiellem Anstieg 64
44 Aufbau Flask-Server 66
45 Komponentenstruktur 71
46 Ausgewählter Tab 72
47 Navbar Viewer & Admin 73
48 Navbar SuperUser 73
49 Dashboard 73
50 API-Schnittstellen 74

Quellcodeverzeichnis

1 Installation von Chart.js . . . 5
2 Einbindung eines Canvas-Elements . . . 5
3 Globales Styling . . . 7
4 Globale SCSS Variablen . . . 7
5 Variablen Import . . . 7
6 Komponenten Styling . . . 8
7 User Model . . . 13
8 MSSQL-Initialisierungsskript . . . 22
9 TimescaleDB-Initialisierungsskript . . . 24
10 Anlegen eines Threads . . . 47
11 Zielfunktion des Threads . . . 47
12 Einlesen einer CSV-Datei mittels Pandas . . . 48
13 herkömmliche Bearbeitung eines Arrays . . . 49
14 Bearbeitung eines Arrays mit Numpy Vektorisierung . . . 49
15 Installation von Ngx-Scrollbar . . . 55
16 Einbingung der Scrollbar . . . 55
17 URL des faststream-Endpunkts . . . 61
18 Auslesen des JPEG Kommentars . . . 61
19 Visualisierung mittels Bokeh . . . 62
20 Parameter zur Datengenerierung . . . 63
21 Generierung einer Safe Sequenz mit kleiner Erhebung . . . 63
22 Generierung einer riskanten Sequenz mit exponentiellem Anstieg . . . 64
23 Aufbau des Modells . . . 65
24 Modelltraining - abstrakte Darstellung . . . 65
25 Überprüfung eines Requests . . . 66
26 Endpunkte des Flask-Servers - abstrakte Darstellung . . . 66
27 Startverhalten des Servers - abstrakt Darstellung . . . 67
28 Kamera Manager - abstrakte Darstellung . . . 67
29 Kamera Worker Ausschnitt 1 - abstrakte Darstellung . . . 68
30 Kamera Worker Ausschnitt 2 - abstrakte Darstellung . . . 69
31 Kamera Worker Ausschnitt 3 - abstrakte Darstellung . . . 69
32 Kamera Worker Ausschnitt 4 - abstrakte Darstellung . . . 70
33 config-file . . . 70
34 Variablen Import . . . 71
35 Routen Definition . . . 71
36 Routen Definition . . . 72
37 Navbar Logik . . . 72
38 Ausgewählter Tab . . . 72
39 Einbindung Navbar . . . 73
40 Laden der User-Rolle . . . 73
41 Guard . . . 74
42 Ausgewählter Tab . . . 75
43 Camera Service . . . 75

44 Role Service . . . 76
45 Dashboard Temperature Service Einbindung . . . 77
46 Temperature Service . . . 77
47 Encryption Service . . . 77
48 Pflichtfeldpruefung . . . 78
49 Fehlerfeedback . . . 78
50 Enter-Handling . . . 78
51 Eingabe triggert Vorschau . . . 79
52 Vorschau-Erfolg schaltet Submit frei . . . 79
53 Countdown im Dashboard . . . 80
54 Kamera Hinzufügen . . . 80
55 Kamera-Liste filtern . . . 80
56 Kamera umbenennen . . . 81
57 Kamera ins Dashboard übernehmen . . . 81
58 Kamera löschen . . . 81
59 Live-Temperatur laden . . . 82
60 Historische Ansicht öffnen . . . 82
61 Einfärbung der Chart-Kurve . . . 83
62 Alarmstatus umschalten . . . 83
63 Linse umschalten . . . 84
64 Kamera vom Dashboard entfernen . . . 84
65 Elemente Selektor . . . 84
66 Rollenanfrage stellen . . . 85
67 User-Rolle ändern . . . 85
68 Rollen-Anfrage bearbeiten . . . 85
69 Backend-Fehler behandeln . . . 86
70 Frontend-Fehler setzen . . . 86
71 Fehlermeldung im Template . . . 86
72 Authentifizierung . . . 86
73 Kamera-Feed Proxy . . . 86
74 Dashboard Live-Feed via Proxy . . . 87
75 Camera List Vorschau via Proxy . . . 87
76 Add Camera Vorschau via Proxy . . . 87
77 Imports . . . 88
78 Initialisieren und Konfigurieren von Swagger/Middleware/Root-Route label . . . 89
79 Mounten der Router . . . 90
80 Initialisierungsfunktion von app.ts . . . 90
81 Cleanup-Job direkt nach Start . . . 91
82 Beispiel: Cleanup-Funktion zum Aufräumen abgelaufener Sessions . . . 92
83 Beispiel: Jährlicher Cron Job zum Aufräumen unverlinkter Kameras . . . 92
84 authorizeRole-Workflow . . . 95
85 Beispiel-Route: Nur Admin und Super User erlaubt . . . 95
86 Beispiel: Model des Users . . . 96
87 getAllUsers . . . 97
88 getUser . . . 97
89 addUser . . . 97
90 UpdateUserInput-Interface . . . 98
91 updateUser . . . 98
92 updateUserRole . . . 98
93 deleteUser . . . 99

94 auth.service.ts 99
95 jwt-config.ts 100
96 Tokens und Session werden erstellt und persistiert 101
97 Passwortvergleich 102
98 Session-Model 103
99 trimSessionsForUser 104
100 role-request.model.ts 104
101 getPendingRoleRequestsWithUsernames 104
102 updateRoleRequestStatus 105
103 createRoleRequestIfNone 105
104 Camera-Model 106
105 User_Camera-Model 106
106 Entschlüsselung des Kamerapassworts 107
107 TemperatureData-Model 109