CARL VON OSSIETZKY universität OLDENBURG

FORSCHUNGSPROJEKT

Entwicklung und Evaluation einer lokalen REST-API

von

Kian Lütke

Kian Lütke
Matr.-Nr.: 5417637
Kranbergstr. 6
26123 Oldenburg
E-Mail: kian.luetke@gmail.com

Prüfer: Prof. Dr. Jürgen Sauer

Abgabe: 01.05.2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface. 2, 4, 7, 20, 24, 31, 32, 34

BEM Bock Element Modifier. 21, 34

CRLF Carriage Return Line Feed. 6
CRUD Create, Read, Update and Delete. 4
CSS Cascading Style Sheets. 10, 12, 21, 34
CSV Character Separated Values. 4, 6, 8, 13, 27, 28, 33

HTML Hypertext Markup Language. 10, 21
HTTP Hypertext Transfer Protocol. 5, 6, 12, 13, 15, 29, 31

JSON JavaScript Object Notation. 5, 7, 8, 15, 16, 17, 21, 31, 35

LAMP Linux Apache MySQL PHP. 10

MVC Model View Controller. 11, 34

NPM Node Package Manager. 22

REST Representational State Transfer. 4, 34
REST-API Representational State Transfer Application Programming Interface. 1, 4, 5, 6, 9, 11, 14, 20, 35

SQL Structured Query Language. 17

URI Uniform Resource Identifier. 5
URL Uniform Resource Locator. 18, 20, 29

VM Virtuelle Maschine. 22

1 Einleitung

1.1 Projektbeschreibung und Motivation

Dieses Forschungsprojekt umfasst die Konzeption und Entwicklung einer lokal verfügbaren *Representational State Transfer Application Programming Interface* (REST-API). Dabei steht die Simulation eines oder mehrerer Serverendpunkte zur Abfrage von normalerweise entfernten Ressourcen im Mittelpunkt. Die Realisierung dieses Projektes erfordert die Auswahl geeigneter Technologien, die Analyse der Anforderungen sowie das Design und die Entwicklung der Software. Idealerweiser soll die Software so konzipiert werden, dass ein weiteres Projekt an diese Arbeit anknüpfen kann, um einen beispielhaften Anwendungszweck zu illustrieren.

Ein Projekt wie dieses, wird zum Beispiel bei einer Präsentation benötigt, in welcher eine unzureichende Internetverbindung zu erwarten ist. So können Beispieldaten in wiedergegeben werden ohne auf einen entfernten Server zuzugreifen. Desweiteren ist in nahezu allen mittleren bis großen Unternehmen eine restriktive Internetnutzung Standard. Das heißt, dass es während der Entwicklung von neuen Projekten nicht unbedingt möglich ist, auf entfernte Ressourcen zuzugreifen [Hun98]. Auch hier kann eine lokale REST-API sinnvoll sein, um mit Beispieldaten arbeiten zu können.
Mock-Ups von Designern können mit Hilfe dieser Software ebenfalls mit generischen Daten versorgt werden, um alle möglichen Ausprägungen in ihren Präsentationen einzuschließen.

1.2 Vorgehensweise

Bei der Entwicklung der Software wird sich an dem Buch *Software Engineering* von *Ian Sommerville* orientiert [Som16]. Auf die Anforderungsanalyse wird dabei in Kapitel 2 eingegangen, welches ebenfalls die System- und Architekturmodellierung umfasst. Die aus diesem Kapitel entstehenden Resultate, werden anschließend in Kapitel 3 in die Praxis umgesetzt. Hier wird nun ein vollständiges Programm entwickelt, welches sowohl die *Application Programming Interface* (API) als auch eine Benutzeroberfläche zur Anpassung der Endpunkte zur Verfügung stellt.

Nach erfolgter Implementierung stellt das Kapitel 4 ein Handbuch dar, welches die Nutzung der Oberfläche und der API illustriert. Im letzten Kapitel wird die Umsetzung überprüft und mit den Anforderungen verglichen. Ein Ausblickt gibt Aufschluss über mögliche Verbesserungen oder Erweiterungen. Die Entwicklung des Programms geschiet mit der Entwicklerumgebung Visual Studio Code[1]. Die schriftliche Ausarbeitung wird in Latex geschrieben und der Quellcode der Abgabe hinzugefügt.

[1] https://code.visualstudio.com/

2 Entwurf

2.1 Anforderungsanalyse

Die Anforderungsanalyse eines Systems oder Programms besteht aus den Beschreibungen der zu leistenden Funktionen und den bestehenden Beschränkungen, welchen das System unterliegt. Dabei wird in *Benutzeranforderungen* und *Systemanforderungen* unterschieden. Ersteres wird dabei in Diagrammen und Listen dargestellt, während letzteres eine detailliertere Beschreibung der Software-Funktionen umfasst [Som16]. In diesem Zusammenhang werden zunächst grobe Anforderungen gesammelt, um anschließend eine genauere Beschreibung darzulegen.
Die Anforderungen werden zudem in funktionale und nicht-funktionale Anforderungen unterteilt [Som16]:

- *Funktionale Anforderungen* stehen für das Verhalten der Software auf zum Beispiel Benutzereingaben oder in diesem Fall Abfragen an einen Server
- *Nicht-funktionale Anforderungen* sind beispielsweise Beschränkungen, welche durch bestehende Standards entstehen.

Daraus lässt sich schließen, dass funktionale Anforderungen direkt mit der Funktion des Systems in Verbindung gebracht werden können, während nicht funktionale Anforderungen die technischen Einschränkungen beschreiben. Die folgende Tabelle illustriert die Benutzeranforderungen und teilt sie in die aufgezeigten Sparten ein:

Funktional	Nicht-funktional
Oberfläche zur Konfiguration der REST-API • Manuelles Hinzufügen von Ressourcen • Hinzufügen von Ressourcen durch *Character Separated Values* (CSV) Datei • Visuelle Darstellung der Endpunkte	Betriebssystemunabhängiges Ausführen der Software
Persistente Datenspeicherung	Operiert ohne Internetverbindung
	REST-Konform

Tabelle 2.1: Benutzeranforderungen

Systemanforderungen

Das System muss auf den bekannten Betriebssystemen *Windows*, *Mac OS* sowie *Linux* lauffähig sein und ohne Internetverbindung gestartet, genutzt und konfiguriert werden können. Das Starten des Programms führt zur ausführung eines Servers, welcher die API und die Benutzeroberfläche bereitstellt. Des Weiteren muss das System *Representational State Transfer* (REST)-Komform sein. Dies bedeutet, es unterliegt einer Reihe von inoffiziellen Regeln bzw. *Best-Practises*, welche im Folgenden aufgelistet werden [Mas12]:

Request Methoden Abfragen oder auch *Requests* werden normgerecht verwendet. Es gibt gewisse Semantiken, die hier Verwendung finden und nach dem *Create, Read, Update and Delete* (CRUD) Prinzip auf die REST-API angewendet werden. [Net20a] [Mas12]:

- **GET** Anzeigen von Ressourcen; wobei ohne Parameter alle Einträge abgerufen werden. (Read)
  Beispiel: *GET http://api.localhost/studenten*
  Beispiel: *GET http://api.localhost/studenten?id=1*
- **DELETE** Entfernen einer Ressource (Delete)
  Beispiel: *DELETE http://api.localhost/studenten?id=1*
- **PUT**/**POST** Hinzufügen/Ändern einer Ressource (Create, Update)

Beispiel: *PUT http://api.localhost/studenten*

Beispiel: *POST http://api.localhost/studenten?id=1*

JavaScript Object Notation (JSON) formatierter Anfragebody:

```
{
    name: "Jordan",
    last_name: "Riley",
    age: "23",
    subject: "Wirtschaftsinformatik"
}
```

URI Format Generell sollte das *Uniform Resource Identifier* (URI) Format, welches den Pfad zum Zugriff auf eine Ressource darstellt, verwendet werden. Dieser wurde 2005 unter dem Standard *RFC 3986* von der *Network Working Group* definiert [Net05]:

```
URI = scheme "://" authority "/" path [ "?" query ] [ "#" fragment ]
```

Es gibt eine Reihe von Regeln, welche bei der Konzeption einer REST-API bedacht werden sollten [Mas12]:

- **Vorwärts Slash (/)** indiziert hierarchische Ebenen.
  Beispiel: *http://api.localhost/personen/studenten*
- **Ein angestelltes Slash** wird am Ende eines URI nicht verwendet.
  Beispiel: *http://api.localhost/studenten/ <-*
- **Bindestriche** werden zur Lesbarkeit verwendet.
  Beispiel: *http://api.localhost/personen/studenten-mit-abschluss*
- **Dateiendungen** sind nicht mit einzuschließen.
  Beispiel: *http://api.localhost/dokumente/lehrplan.pdf*
- **Suchanfragen** werden nicht in den URI mit einbezogen, sondern dem *Hypertext Transfer Protocol* (HTTP)-Standard gemäß übergeben.
  Beispiel (Falsch): *http://api.localhost/studenten/getById/2*
  Beispiel (Richtig): *http://api.localhost/studenten?id=2*

Status Codes HTTP Status Codes informieren den Client über das Ergebnis der eingangs gestellten Abfrage. [Net20a] Diese sind ebenfalls semantisch korrekt auf die REST-API anzuwenden:

- **200 OK** ist ein allgemeiner Code, welcher über eine erfolgreiche Abfrage informiert.
- **201 CREATED** informiert über die erfolgreiche Erstellung einer Ressource.
- **404 NOT FOUND** gibt Auskunft über nicht zur Verfügung stehende Ressourcen.
- **500 INTERNAL SERVER ERROR** sagt aus, das ein serverseitiger Fehler aufgetreten ist.

CSV Datei Neben den Anforderungen an die REST-API, ist es von Bedeutung festzulegen, in welchem Format die CSV-Datei sein sollte, um vom Server akzeptiert zu werden. Dem Standard RFC 4180 zufolge, werden verschiedene Werte durch Kommata getrennt. Reihen hingegen werden durch einen Zeilenumbruch voneinander separiert: *Carriage Return Line Feed* (CRLF). Die erste Zeile der Datei bildet die Kopfdaten oder auch Spaltennamen [Net20b].

```
field_name,field_name,field_name CRLF
aaa,bbb,ccc CRLF
zzz,yyy,xxx CRLF
```

2.2 Systemmodellierung

Die Systemmodellierung befasst sich primär mit der Überführung der Anforderungen in abstrakte Modelle. In diesem Kapitel wird analysiert, wie sich das System auf bestimmte Benutzereingaben verhält und welche Anwendungsszenarien zur Verfügung stehen. Dies wird Anhand von Use-Case- und Sequenzdiagrammen aufgezeigt. [Som16]

Verhaltensanalyse In diesem Szenario besteht das System aus drei Teilnehmern; dem Server *(Backend)*, der Konfigurationsoberfläche (*Frontend*) und dem Anwender

(*User*). Im ersten Fall wird dargestellt, welche Aktionen mit dem Hinzufügen von Ressourcen einhergehen (Abbildung 2.1). Zunächst ist es dem Anwender möglich eine CSV-Datei hochzuladen oder eine Tabelle mit Daten manuell zu erstellen. Nach dem Bestätigen, werden die in der Tabelle enthaltenen Daten an den Server übertragen. Dieser legt seinerseit eine Datenbanktabelle mit entsprechenden Namen, Spalten und Spaltendaten an. Auch die API-Endpunkte werden in einer entsprechenden Konfigurationsdatei festgelegt. Dabei werden alle in Abschnitt 2.1 Methoden (GET, PUT/POST, DELETE) implementiert, um das Manipulieren der Daten im Anschluss zu ermöglichen.
Nach erfolgreicher Erstellung wird das Frontend informiert. Dieses wiederum aktualisiert die Anzeige und stellt dem Anwender die API-Endpunkte grafisch dar. Das

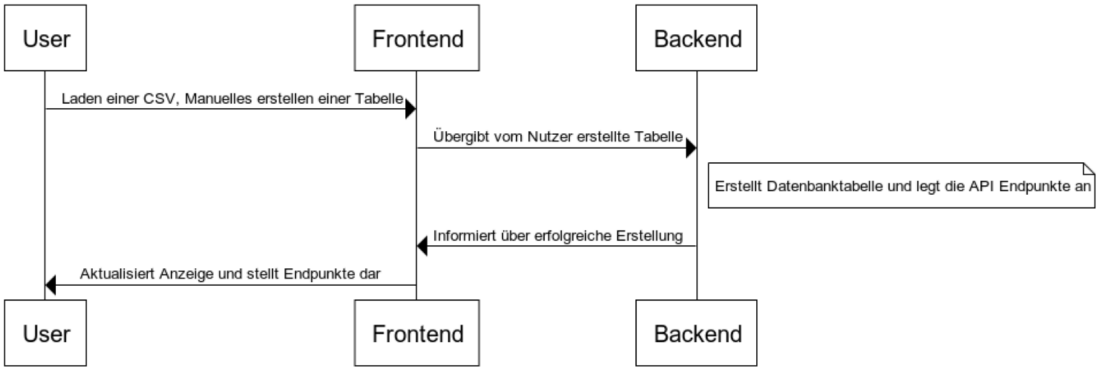

Abb. 2.1: Hinzufügen von Ressourcen

zweite Szenario bezieht sich auf die Abfrage einer Ressource. Zunächst wird hierbei eine *GET*-Abfrage an den Server gestellt.

```
GET http://localhost:3000/api/users?id=1
```

Erwartungsgemäß sollte die Antwort des Servers ein JSON-Objekt sein, welches die gespeicherten Daten des Users mit der ID 1 enthält. Nach dem Eingang der Anfrage prüft das Backend zunächst, ob die Ressource vorhanden ist. Ist dies nicht der Fall, antwortet der Server mit dem HTTP Status-Code *404 NOT FOUND*. Dies indiziert das Nichtvorhandensein einer angefragten Ressource. Ist sie jedoch

vorhanden, ruft der Server die zugrundeliegenden Daten aus der Datenbank ab und gibt eine entsprechende JSON-formatierte Nachricht zurück. Darin enthalten sind mehrere Attribute. Der Status, eine Nachricht (message), in der auch eventuelle Fehlermeldungen an den Client zurückgegeben werden können und die eigentlichen Daten (data). Daraus lässt sich also schließen, dass der Anwender auf der

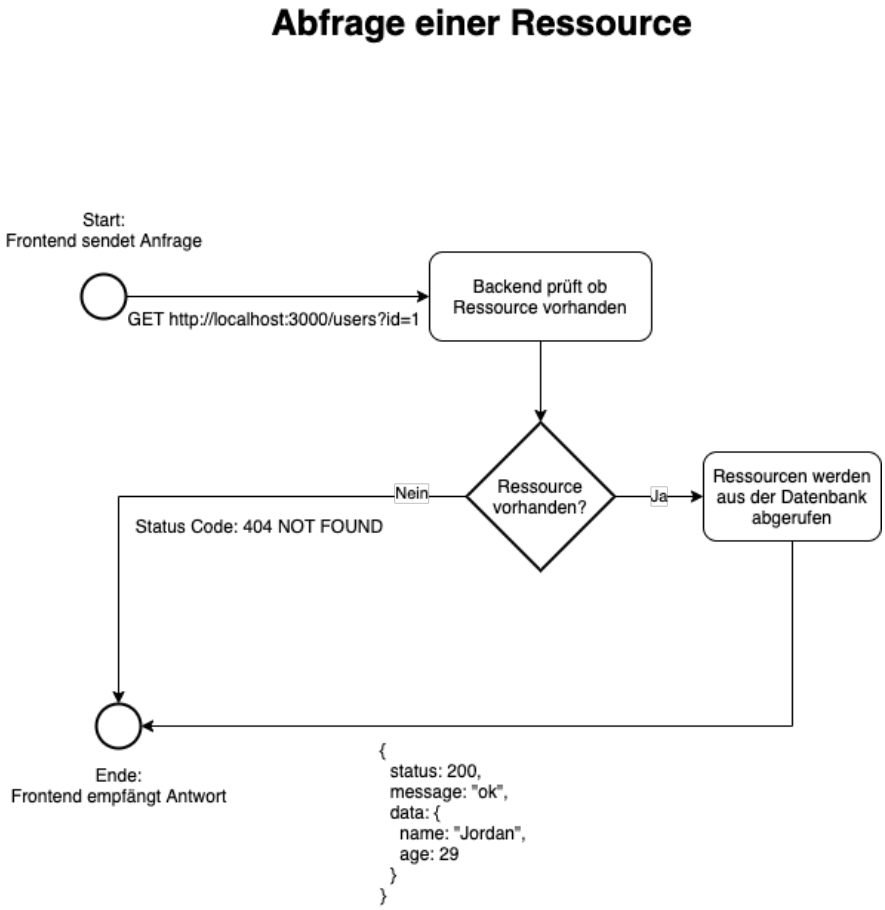

Abb. 2.2: Abfrage einer Ressource

Konfigurationsoberfläche die Möglichkeit hat Daten entweder manuell in eine Tabelle einzutragen oder eine eigens erstellte CSV-Datei hochzuladen.
Das Frontend muss mit dem Backend kommunizieren, um Tabellen anzulegen oder zu entfernen. Dies wird über eigene Endpunkte zur Administration geschehen.

```
PUT http://localhost:3000/api/admin/table
```

```
DELETE http://localhost:3000/api/admin/table?name=users
```

Über den *PUT*-Request können Tabellen neu hinzugefügt oder überschrieben werden. Über den *DELETE*-Request werden Tabellen hingegen gelöscht. Ferner müssen Endpunkte zur Manipulation der Daten bereitgestellt werden. Da diese immer einheitlich sind, genügt es die Tabellennamen in einer Datenbank zu speichern und somit auf Validität der Endpunkte bei Abfragen zu prüfen.

2.3 Architekturdesign

Dieses Kapitel befasst sich zunächst mit dem groben Aufbau der Anwendung. Hierbei wird skizziert, wie Frontend, Backend und Datenbank in Relation zueinander stehen. Der zweite Teil dieses Kapitels befasst sich mit der verwendeten Technologie zur Programmierung des Frontends, Backends und der verwendeten Datenspeicherungsmethode.

Grober Entwurf

Der erste Entwurf des Aufbaus (Abbildung 2.3 - Icons von Flaticon[2]) ist in vier Teile unterteilt: Backend, Frontend, Client, Datenbank. Der Server (Backend) bildet dabei die zentrale Vermittlungsstelle. Hier werden eingehende Anfragen von der Konfigurationsoberfläche und dem Client (REST-API Nutzer) bearbeitet und beantwortet. Des Weiteren agiert der Server mit einer Datenbank, um neue Tabellen zu speichern, bestehende zu überschreiben oder zu löschen.

Verwendete Technologien

Zur Entwicklung des gesamten Projekts müssen in diesem Abschnitt die richtigen Technologien ausgewählt werden. Dazu gehören zum Beispiel die Programmiersprachen des Frontends und des Backends. Um die persistente

[2] https://www.freepikcompany.com/legal

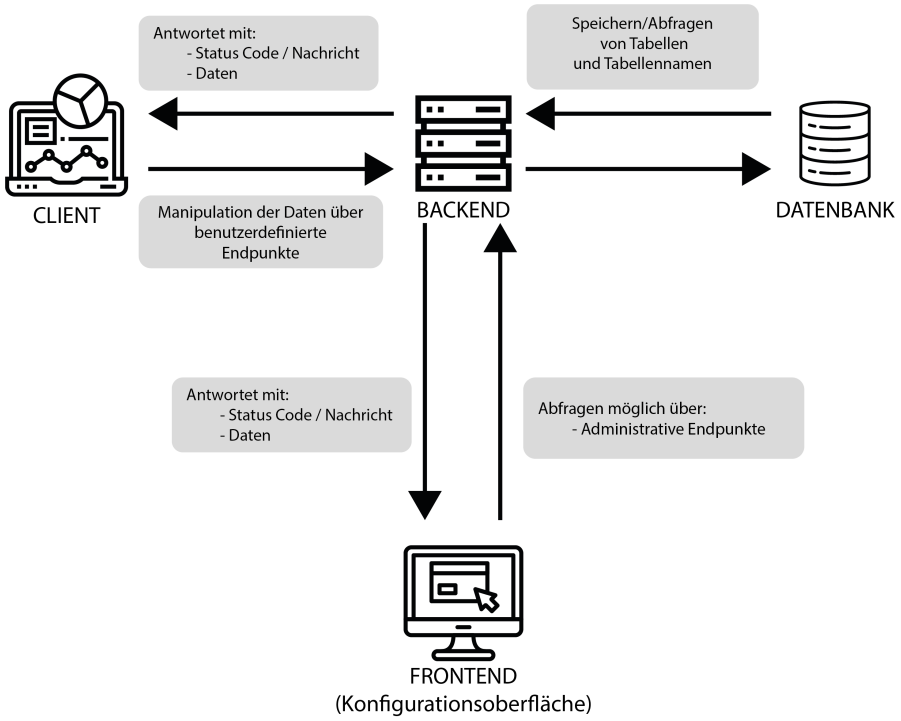

Abb. 2.3: Grobkonzept

Datenspeicherung zu ermöglichen, wird auch eine geeignete Datenbank ausgewählt. Klassische Lösungen basieren auf dem sogenannten *Linux Apache MySQL PHP* (LAMP)-Stack. Hierbei müsste lokal ein Apache-Server[3] gestartet werden, welcher PHP-Dateien[4] bereitstellt. Diese wiederum können mit einer MySQL[5] Datenbank kommunizieren und die benötigten Daten bereitstellen. Zur Gestaltung der Webseiten kommen *Hypertext Markup Language* (HTML) und *Cascading Style Sheets* (CSS) zum Einsatz. Während HTML die reine Auszeichnungssprache ist und das Grundgerüst von Komponenten darstellt, welche innerhalb des Browsers erzeugt werden, ist letzteres eine Sprache, um die Komponenten in zum Beispiel Farbe und Form zu gestalten. Die funktionale Programmiersprache *JavaScript* wird hier nur dazu verwendet, um Webseiten dynamischer zu gestalten. So ist beispielsweise das Auslesen von Eingabefeldern oder das Auswerten eines Knopfdrucks möglich [Ger06].

[3] https://httpd.apache.org/
[4] https://www.php.net/manual/de/intro-whatis.php
[5] https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html

Die Programmiersprache JavaScript hat sich im laufe der Zeit sehr weit entwickelt und findet heutzutage auch außerhalb der Browserumgebung Verwendung. Die Sprache wird beispielsweise auch serverseitig eingesetzt und durch eine Laufzeitumgebung wie *node.js*[6] für serverseitige Anwendungsszenarien eingesetzt. [RBM16] Gleichzeitig ließe sich die Konfigurationsoberfläche ebenfalls mit node.js bereitstellen. Der Vorteil Front- und Backend in einer gemeinsamen Sprache zu programmieren legt nahe node.js für dieses Projekt einzusetzen.
Offen bleibt die Frage, wie Daten innerhalb des Systems persistiert werden. Wie bereits erwähnt, ist eine Möglichkeit die Nutzung einer relationalen *MySQL* Datenbank. Jedoch wird hierfür ein weiterer ressourcen-intensiver Server benötigt [Ger06]. Für einen solchen, eher kleinen Datenbestand, lässt sich eine *SQLite*-Datenbank in Betracht ziehen. Im Gegensatz zur herkömmlichen MySQL-Datenbank, werden Daten in einer Datei abgelegt und zur Laufzeit aus dieser gelesen respektive geschrieben. Trotz der serverlosen Datenbankarchitektur ermöglicht SQLite die meisten Features, welche MySQL bietet.[7]

Systemarchitektur

Ein nicht von der Hand zu weisender Faktor bei der Entwicklung von Softwareprojekten, ist der zugrundeliegende Aufbau einer Software. Bei modernen Webanwendungen wird auf das sehr weit verbreitete *Model View Controller* (MVC) Muster zurückgegriffen. Dabei wird das System in mehrere Schichten unterteilt, welche verschiedene Bereiche voneinander trennen. Bei einer in node.js geschriebenen REST-API sind das typischerweise:

- Model: Datenmodell; Zugriff auf Datenbank (Lesen, Schreiben); Umwandlung in JS-Objekte

[6] https://nodejs.org/en/about/
[7] https://www.sqlite.org/about.html

- Controller: Verarbeiten, validieren und beantworten der eingehenden HTTP-Anfragen.
- Service: Business-Logik, hier ist der logische Programmcode implementiert
- Tests: Modultests, um einzelne Bereiche zu testen.
- Sonstige: util - gemeinsame Funktionen; constants - gemeinsame Konstanten

In Abbildung 2.4 ist der Aufbau des Projekts im Detail erkennbar. Der Ordner *node_modules* beinhaltet lediglich Module, welche von node.js genutzt werden. Die Konfigurationsoberfläche (View) ist ebenfalls in dieser Struktur, in Form des *Public*-Ordners, vorhanden. Der Aufbau des Frontend-Ordners ist ebenfalls Modular unterteilt. So befindet sich CSS in dem dafür angelegten Ordner. *Assets*, also Bilddatein, Schriftarten oder sonstige Dateien, werden im gleichnamigen Ordner abgelegt. Die JavaScript-Dateien befinden sich ebenfalls in einem Unterordner.
Dank der Einführung von JavaScript-Modulen im Browser mit dem Standard *ECMA-Script 2016* wird hier auch ein modularer Aufbau ermöglicht. Im Gegensatz zum normalen Webseiten Aufbau, kann in die Webseite ein einziges Modul importiert werden (*main.js*), welches auf Funktionen in den einzelnen Untermodulen zugreift. Dies ermöglicht einen strukturierten und lesbareren Aufbau. [Moz20]

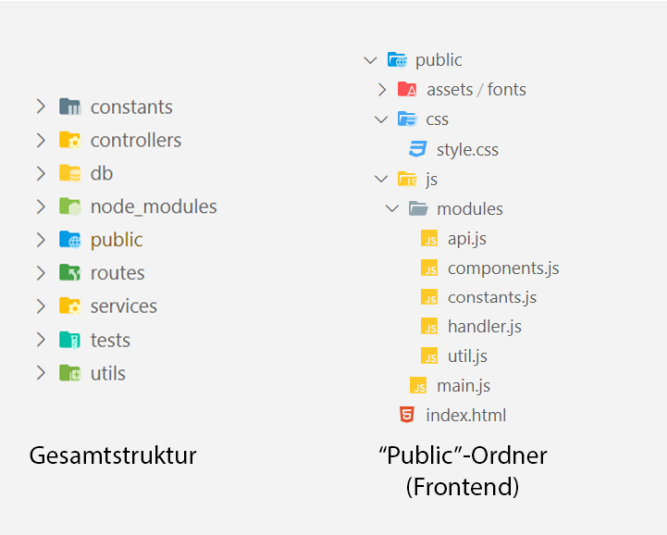

Abb. 2.4: Projektstruktur

3 Implementierung

3.1 Design Prototyp

Um der Konfigurationsoberfläche ein Aussehen verleihen zu können, wird zunächst ein Design-Prototyp erstellt. Hierzu werden alle Komponenten aufgelistet, welche in diesem Projekt Verwendung finden.

- Titelleiste mit Menüunterpunkt zum Hinzufügen neuer Endpunkte
- Aufklappbare Endpunktliste. Beim Aufklappen sollen Informationen zum Endpunkte ersichtlich sein.
- Neuer-Endpunkt/Endpunkt bearbeiten Dialog

Der erste Prototyp wird mit Adobe Illustrator[8] erstellt. Das Design, wird absichtlich schlicht und übersichtlich gehalten.
Im oberen Teil lässt sich die Titelleiste erkennen, welche auch als Menü fungiert. So lässt sich mit dem Klick auf den rechten Menüpunkt zum Beispiel ein Fenster öffnen. Hier kann dann ein neuer Endpunkt manuell erstellt oder mittels CSV-Datei angelegt werden. Nach erfolgreichem Anlegen eines neuen Endpunkts erscheint dieser wiederrum in der Liste im Hauptmenü. Diese Einträge lassen sich ausklappen, sodass sichtbar wird, wie diese Endpunkte zu verwenden sind. Entsprechende HTTP-Methoden werden hier ersichtlich sein. Zusätzlich wären in diesem Teil auch noch weitere Bedienelemente denkbar, welche das Löschen oder Anzeigen der Endpunkte ermöglicht.

[8] https://www.adobe.com/de/products/illustrator.html

Die Anzeige der Endpunkte kann zum Beispiel in Tabellenform erfolgen. Die konkrete Implementierung wird im Kapitel 3.3 illustriert.

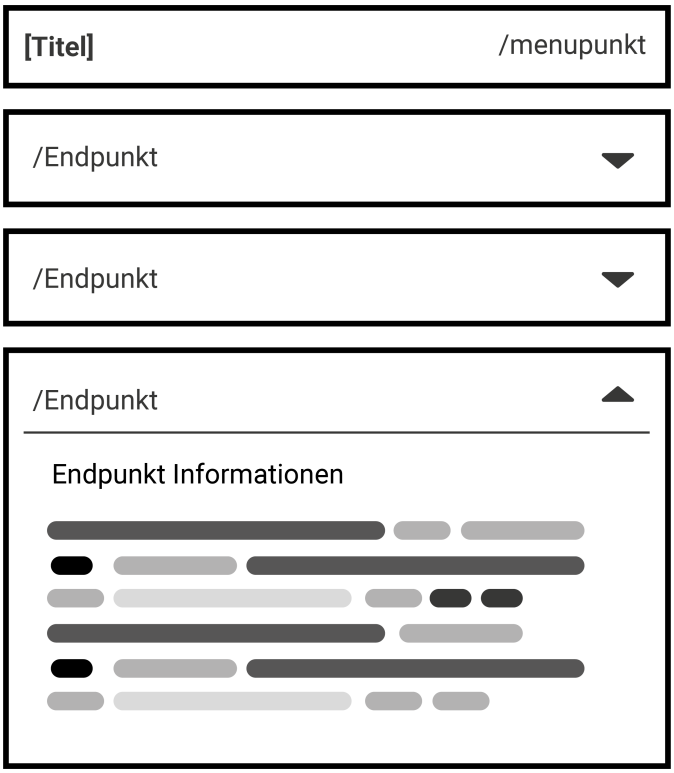

Abb. 3.1: Design-Prototyp

3.2 Backend

3.2.1 Express.js

Das sehr bekannte Framework *express.js* wird verwendet, um Endpunkte der REST-API bereitzustellen. Auch das im nächsten Kapitel zu entwickelnde Frontend kann mittels dieses Frameworks bereitgestellt werden. Es bietet einige rubuste Module für die Entwicklung von Webapplikationen. In dem Buch *REST API Development with Node.js* wird ein Generator verwendet, welcher das initiale Aufsetzen des Projektes und der Ordnerstrukturen stark vereinfacht. Der sogenannte *express-generator* füllt ein leeres

Projekt mit Ordnern, Dateien und beispielhaftem Code [Dog18]. Auf diesen Generator wird jedoch verzichtet, um die Entwicklung des Projektes genauer Untersuchen zu können und selbst die geeignete Ordnerstruktur anzulegen.
Die Struktur wurde bereits im Kapitel 2.3 festgelegt. Auf dieser aufbauend werden die einzelnen Komponenten im Folgendem näher betrachtet.

3.2.2 Controller

Controller werden in diesem Projekt verwendet, um eingehende HTTP-Anfragen zu validieren und die HTTP-Schicht von der Business-Logik zu trennen. Wenn eingehende Anfragen nicht im richtigen Format vorliegen werden sie an dieser Stelle direkt mit einem entsprechenden Status-Code und eine Mitteilung beantwortet.

```
const createTable = async (req, res, next) => {
  const { table } = req.body;
  try {
    if (typeof table === 'undefined') {
      // Requestbody hat falsches Format
      return res
        .status(400)
        .json({ status: 400, message: MESSAGE_FORMAT_ERROR }); // BAD REQUEST
    }
    const responseBody = await adminService.createTable(table);
    res.status(201).json(responseBody);
    next();
  } catch (e) {
    res.sendStatus(500) && next(e);
  }
};
```

Abb. 3.2: createTable Methode des Admin-Controllers

In der Abbildung 3.2 lässt sich eine solche Prüfung erkennen. Zunächst wird mit

```
const { table } = req.body;
```

aus dem, in der Anfrage übergebenem, JSON-Body das Objekt *table* ausgelesen. im nächsten Schritt wird dann geprüft, ob dieses Objekt definiert ist. Im Negativfall wird direkt eine Antwort mit dem Status-Code 400 generiert. Dieser steht für *BAD REQUEST*; also fehlerhafte Anfrage. Ist das Objekt *table* jedoch definiert, wird es zur weiteren Verarbeitung an die Business-Logik weitergereicht.

3.2.3 Services

Im Ordner *Services* befindet sich die Business-Logik. Eingehende Anfragen werden aufbereitet, an die Datenbank gesendet und im JSON-Format wieder an die *Controller*-Schicht zurückgegeben.

```
const createTable = async (table) => {
  try {
    // Tabelle anlegen
    await adminDb.createTable(table);
    // Daten in Tabelle schreiben
    await adminDb.insertDataIntoTable(table);

    // Response Body erstellen und zurückgeben
    const responseBody = {};
    responseBody.status = 201; // CREATED
    responseBody.message = MESSAGE_CREATED;
    responseBody.data = await adminDb.getTableNames();

    return responseBody;
  } catch (e) {
    throw new Error(e.message);
  }
};
```

Abb. 3.3: createTable Methode des Admin-Services

In Abbildung 3.3 wird das vorherige Beispiel zum Erstellen einer Tabelle wieder aufgegriffen. Die Funktion nimmt eine Tabelle im folgenden Format entgegen:

```
{
    table: {
        name: "Tabellenname (Pfadname)",
        header: ["Erster Spaltenname", "Zweiter Spaltenname"....],
        data: [
            ["Erste Reihe 1", "Erste Reihe 2"....],
            ["Zweite Reihe 1", "Zweite Reihe 2"....]
        ]
    }
}
```

Diese Struktur wird nicht weiter verändert und über die Modul-Methode *adminDb.createTable(table)* respektive *adminDb.insertDataIntoTable(table)* an die nächste Schicht weitergegeben. Nach erfolgreicher Ausführung wird eine Antwort generiert. Hierzu wird ein neues Objekt erstellt: *responseBody*. Diesem Objekt werden Attribute hinzugefügt.

- responseBody.status: Der zurückzugebene Status-Code (in diesem Fall 201 CREATED)
- responseBody.message: Eine textuelle Nachricht um anzuzeigen, dass die Ressource erfolgreich erstellt wurde.
- responseBody.data: Hier werden noch einmal von der Model-Ebene alle Tabellennamen abgerufen, um sie mit der JSON-Formatierten Antwort an die Controller-Ebene zurückzugeben.

3.2.4 Model

Im Modul *DB* (in diesem Ffalle das Datenmodell) findet die Kommunikation mit der zugrundeliegende *sqlite*-Datenbank statt. Anfragen aus der Services-Schicht werden entgegengenommen, verarbeitet und beantwortet. Um das Beispiel *createTable* weiterhin zu verfolgen, werden aus diesem Modul drei verschiedene Methoden vorgestellt.
Die *createTable*-Methode in Abbildung 3.4 nimmt ein Objekt im schon bekannten Format entgegen. Eine eventuell bestehende Tabelle wird zunächst entfernt. Dadurch kann diese Funktion auch genutzt werden, um änderungen in die Datenbank zu schreiben.
Diese Funktion ist in Abbildung 3.5 zu sehen und ist eine Ummantelung des *Structured Query Language* (SQL)-Statements *DROP TABLE IF EXISTS tabellenname* [Ger06].
Etwas komplexer dagegen ist die Funktion *insertDataIntoTable* (Abbildung 3.6), welche ebenfalls das Objekt *table* entgegennimmt. Diese Funktion besteht aus zwei Unterfunktionen. Die Unterfunktionen *insert* wird für jede anzulegende Tabellen-Zeile von insertMany aufgerufen. Hier wird ein SQL-Statement geformt [Ger06]:

```
const createTable = (table) => {
  // Tabelle löschen falls vorhanden, um neue Daten zu schreiben
  deleteTable(table.name);
  let sqlString = `CREATE TABLE ${table.name}(
    id INTEGER PRIMARY KEY AUTOINCREMENT,`;
    // Über die Header inkrementieren und den sqlString erweitern
  table.header.forEach((colName) => {
    sqlString += `${colName} TEXT,`;
  });
  // abschließendes Komma entfernen und SQL-Befehl schließen
  sqlString = sqlString.substring(0, sqlString.length - 1);
  sqlString += ');';
  const statement = db.prepare(sqlString);
  return statement.run();
};
```

Abb. 3.4: createTable Methode des Admin-Models

```
const deleteTable = (tableName) => {
  const sqlString = `DROP TABLE IF EXISTS ${tableName}`;
  const statement = db.prepare(sqlString);
  return statement.run();
};
```

Abb. 3.5: deleteTable Methode des Admin-Models

```
INSERT INTO tabellenname ('Spaltenname 1', 'Spaltenname 2'....)
VALUES ('Wert 1 Zeile n', 'Wert 2 Zeile n'....);
```

Nach der Formung wird die gebildete Zeichenkette mit der *.run()* Funktion auf der Datenbank ausgeführt. Dieser Vorgang wird so oft wiederholt, wie Zeilen mit übergeben wurden.

3.2.5 Routes, Konstanten und app.js

In diesem Unterkapitel werden die kleineren Module behandelt. Das Modul *Routes* beinhaltet alle Endpunkte des Systems; sowohl die der Konfigurationsoberfläche als auch die generischen Endpunkte des Clients. Wie in Abbildung 3.7 zu Erkennen, wird bei den Client-Routen der Platzhalter * genutzt. Somit wird hier jeder Aufruf der *Uniform Resource Locator* (URL)

```
const insertDataIntoTable = (table) => {
  const insert = (valueString) => {
    const sqlString = `INSERT INTO ${table.name} ('${table.header.join(
      '\',\'',
    )}') VALUES (${valueString});`;
    return db.prepare(sqlString);
  };

  const insertMany = db.transaction((tableData) => {
    tableData.forEach((values) => {
      const valueString = `'${values.join('\',\'')}'`;
      insert(valueString).run();
    });
  });

  return insertMany(table.data);
};
```

Abb. 3.6: insertDataIntoTable Methode des Admin-Models

`https://localhost:3000/api/`

abgefangen und verarbeitet. Auch Suchanfragen, sogenannte *search querys*, werden hier berücksichtigt. Das Auslesen und Formatieren der Anfragen geschieht auf den unteren Ebenen (Controller, Services).

```
// Administrative Routen um Tabellen anzulegen/löschen
router.get('/admin/table', adminController.getTableNames);
router.get('/admin/table/*', adminController.getTable);
router.put('/admin/table', adminController.createTable);
router.delete('/admin/table', adminController.deleteTable);

// Clientrouten mit generischem Inhalt (kann "searchquery" beinhalten)
router.get('/api/*', clientController.getContent);
router.put('/api/*', clientController.createContent);
router.delete('/api/*', clientController.deleteContent);
```

Abb. 3.7: Alle Routen des Systems

Standard-Nachrichten, Ports und der App-Name werden im Modul *Constants* aufbewahrt, somit sind diese global an jeder Stelle des Projekts verfügbar.

Eine weitere wichtige Schlüsseldatei bildet die *app.js*. In dieser Datei wird der *express.js*-Server gestartet und mit den eben angeführtem Routen verbunden. Auch die statische Bereitstellung des Frontends wird hier ausgeführt. Dies wird im Folgenden

Kapitel näher beschrieben.

3.3 Frontend

3.3.1 Bereitstellung und Modularisierung

Das Frontend (die Benutzeroberfläche) mit der die lokale REST-API angepasst wird ist eine reguläre Webseite, welche auf das Backend über die administrativen API Schnittstellen zugreift. Die Ordner-Struktur ist relativ simpel gehalten. So gibt es einen Wurzelordner in dem sich die Hauptseite befindet (index.html) und drei Unterordner - *css*, *js* und *assets*.
Der Ordner *js* beinhaltet JavaScript Dateien, welche die Modularisierung und die Kommunikation mit dem Backend übernehmen. In dessen Unterordner modules befinden sich Module, welche zur Laufzeit geladen werden. Dieses Prinzip ist ein relativ neues JavaScript-Feature und wird von allen modernen Browsern nativ unterstützt.
Durch die Modularisierung müssen nicht mehr alle Code-Teile eines Projekts in einer Datei gespeichert werden, sondern werden dynamisch importiert wenn nötig. [Moz20]
Die Module sind wie folgt unterteilt:

- *api.js*: In diesem Modul findet ausschließlich die Kommunikation mit dem Backend statt. Die administrativen Endpunkte, welche vorher ausgearbeitet wurden, werden hier angesprochen.
- *components.js*: Dieses Modul beherbergt Komponenten, welche dynamisch geladen und mit übergebenen Werten ausgefüllt werden. Dies verspricht einen hohe Wiederverwendbarkeit einzelner Komponenten. So wird beispielsweise eine Eintragskomponente je nach Tabelle mit den verschiedenen Hinweisen ausgefüllt. (Beschreibung der Endpunkte)
- *constants.js*: Hier werden nur Konstanten gespeichert welche innerhalb der Oberfläche benötigt werden, wie Endpunktmethoden, Port oder die URL des Backends.

- *handler.js*: Diese Komponente ist für jede User-Interaktion auf der Oberfläche zuständig. Sie füllt Tabellen oder reagiert auf Knopfdrücke. Auch die Modals werden hier geladen und mit Fehler- oder Erfolgsmeldungen bestückt.
- *util.js*: in *util.js* sind Funktionen untergebracht, welche andere Funktionen unterstützen. Konvertierungen sind ein solcher Anwendungsfall. Dies wird benötigt, um zum Beispiel eine reguläre HTML-Tabelle in das von der API benötigte JSON-Format umzuwandeln.

Die Bereitstellung der Konfigurationsoberfläche erfolgt über den schon vorhandenen Webserver, welcher über *express.js* ausgeführt wird. Erreichbar ist die Seite über einen Webbrowser unter der Adresse:

```
http://localhost:3000
```

In der Konsole wird hierzu eine entsprechende Nachricht mit dem Link ausgegeben.

BEM: Block Element Modifier

Die Anpassung der Website wird mit Hilfe von CSS durchgeführt. Es gibt verschiedene Arten zur Strukturierung einer solchen Datei. Eine Möglichkeit ist die Strukturierung nach dem *Bock Element Modifier* (BEM) Prinzip. Die Idee hinter dieser Namensgebung ist jedes zu stilisierende Element in bestimmte Kategorien aufzuteilen [Vse20, Fin17]:

- Block: Elemente die für sich selbst stehen wie z.B. *header*, *container*, *menu* oder *input*
- Element: Entität die nicht für sich selbst steht aber fest an einen Block gebunden ist wie z.B: *header-title* oder *menu-item*
- Modifier: Diese Attribute ändern die Erscheinung einzelner Elemente wie Farbe oder Größe z.B. *color-dark*, *size-m*

3.4 Bündelung in Applikation

Um die Applikation für verschiedene Betriebssysteme bereitzustellen, müssen die Quelldateien möglichst in ausführbare Dateien gebündelt werden. Idealerweise wird hierbei die Umgebung node.js direkt mit eingeschlossen, um komplizierte Installationsvorgänge zu vermeiden.

Das populärste Werkzeug ist das über *Node Package Manager* (NPM) installierbare Werkzeug *pkg*. Dieses Programm erzeugt automatisch die gewünschten ausführbaren Dateien, sodass nur noch der Ordner mit den Modulen und der mit den statischen Dateien der Konfigurationsoberfläche ausgeliefert werden müssen. Der Quellcode des Backends, sowie eine aktuelle node.js-Version werden in die ausführbaren Dateien eingebunden. [igo20]

Jedoch sind bei der Benutzung in der Praxis Probleme aufgetreten. Da proprietärer Code zur Benutzung der Datenbank auf dem jeweiligen System (welches pkg ausführt) kompiliert wird, ist dieser nicht mehr mit anderen Betriebssystemen kompatibel. Das heißt also ein Paket, welches unter MacOS gepackt wurde, lässt sich auch nur unter MacOS ausführen.

Dies kann umgangen werden durch die Kompilierung auf unterschiedlichen Systemen, sodass drei verschiedene Versionen entstehen; für Windows, für Linux und eine für MacOS. Dafür werden im Rahmen dieses Projektes je eine *Virtuelle Maschine* (VM) installiert mit den Betriebssystemen Windows 10 und Debian 10. In diesen werden dann alle benötigten Werkzeuge installiert, wie node.js, pkg und NPM wie in Abbildung 3.8 ersichtlich.
Nach der erfolgreichen Kompilation auf den verschieden emulierten- und dem Host-Betriebssystem, stehen drei App-Versionen bereit. Ein kurzer Testlauf hat gezeigt, dass diese reibungslos funktionieren.

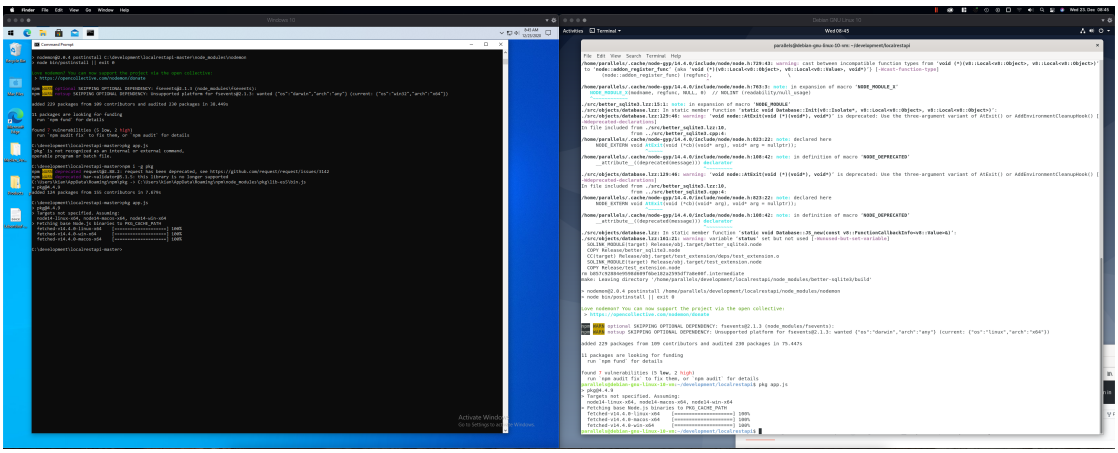

Abb. 3.8: Virtuelle Maschinen zum Kompilieren

4 Handbuch: Benutzung des Programms

4.1 Konfigurationsoberfläche

4.1.1 Anzeigen und ändern eines Endpunktes

Nach dem Aufrufen der Administrationsoberfläche, sind entweder vorhandene Tabellen gelistet oder es wird, bei Nichtvorhandensein von Tabellen, nur die Kopfleiste angezeigt. In Abbildung 4.1 ist bereits ein Endpunkt zu sehen: *artikel*.

Abb. 4.1: Startbildschirm der Administrationsoberfläche

Mit einem Klick auf diesen Endpunkt wird eine Übersicht über die Benutzung angezeigt. Die verschiedene Spalten geben Auskunft über die zu verwendende Methode und dem Pfad. Zudem gibt es eine Spalte *Beschreibung* in der noch einmal beschrieben wird wie die jeweilige Methode zu nutzen ist. Eine genaue Erklärung zur Nutzung der API erfolgt im Abschnitt 4.2.

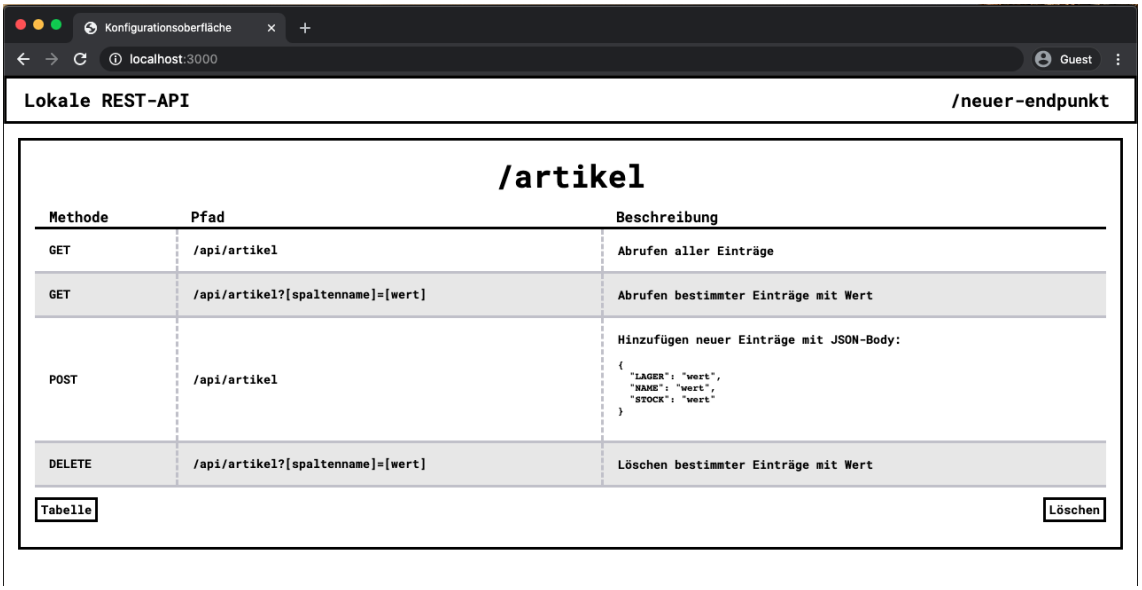

Abb. 4.2: Anzeige eines Endpunktes

Mit einem Klick auf den Button *Tabelle*, öffnet sich ein Modal in dem die dazugehörigen Daten stehen (Abbildung 4.3). Oben links lässt sich die Form der Tabelle einsehen - in diesem Fall 4 Spalten und 4 Zeilen. Durch Ändern der Zahlen können hier beliebig Zeilen hinzugefügt oder entfernt werden.

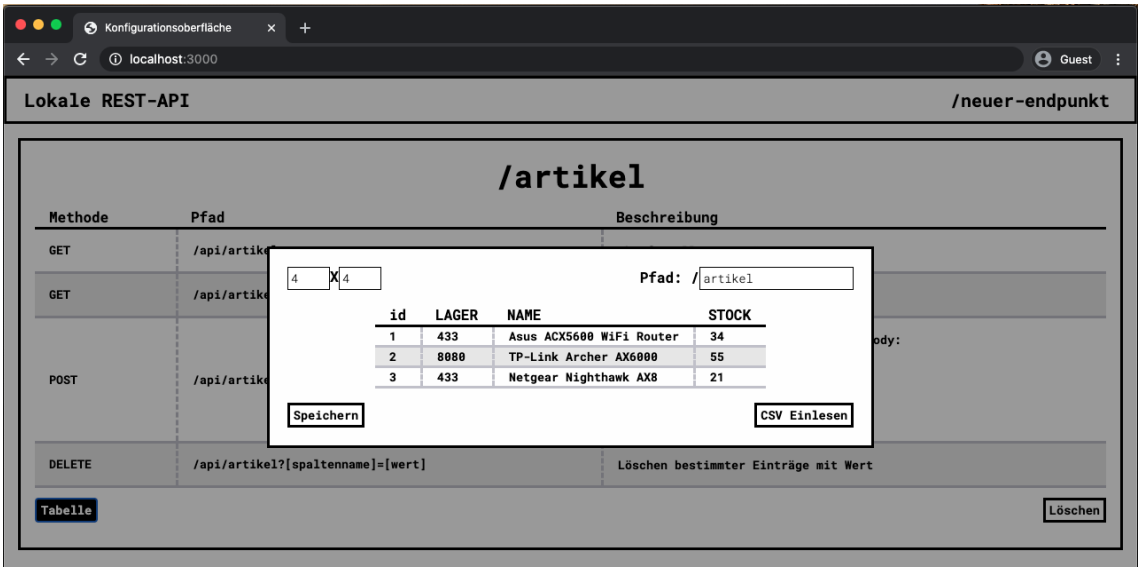

Abb. 4.3: Änderung eines bestehenden Endpunktes

Die Tabelle ist beschreibbar, was bedeutet, dass Daten durch Überschreiben geändert werden können. Um gewünschte Änderungen abschließend zu persistieren, ist eine Bestätigung durch Drücken des *Speichern* Knopfes erforderlich.

4.1.2 Erstellung eines neuen Endpunktes

Um einen neuen Endpunkt anzulegen, wird über den Menüpunkt *neuer-endpunkt* ein Modal geöffnet. Es sind mehrere Möglichkeiten zum Erstellen verfügbar. Die erste ist die manuelle Eingabe: Hierzu wird zunächst das Format der Datentabelle festgelegt, der Pfad des Endpunktes angegeben und anschließend Daten in die Tabelle eingetragen. Eine eindeutige ID ist nicht erforderlich, da diese automatisch bei Speicherung der Tabelle erzeugt wird. Nach erfolgreicher Erstellung wird die Startübersicht inklusive der neu erstellten Tabelle angezeigt.

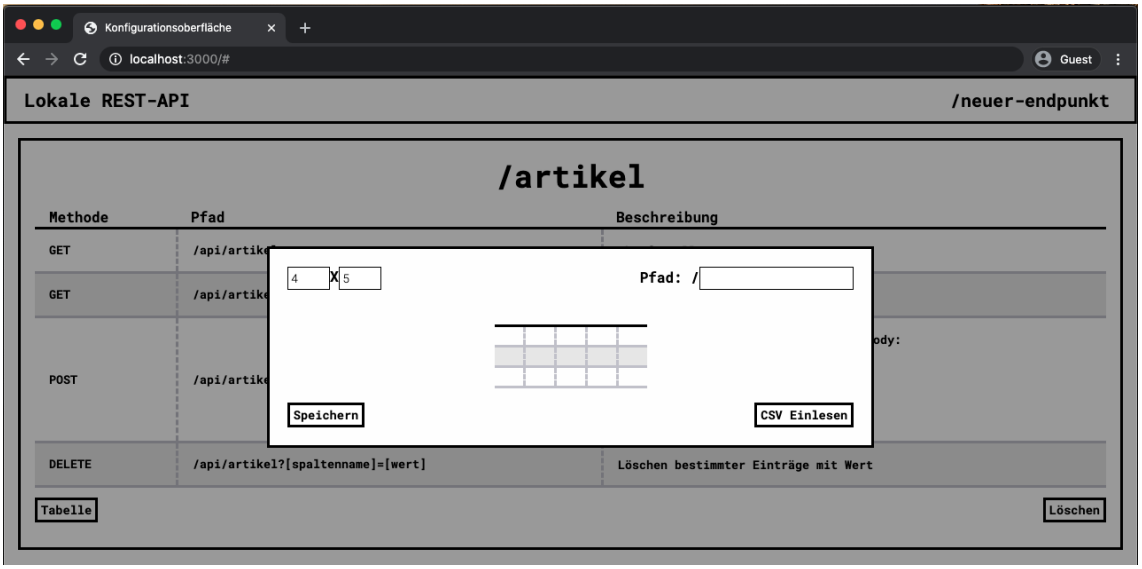

Abb. 4.4: Anlegen eines neuen Endpunktes

Weiterhin ist es möglich eine CSV-Datei einzulesen. Das einzuhaltende Format ist in Abbildung 4.5 abgebildet. Die erste Zeile ist immer die Kopfzeile der Tabelle. Spalten werden durch Kommata voneinander abgegrenzt und eine neue Zeile wird durch einen Zeilenumbruch gekennzeichnet.

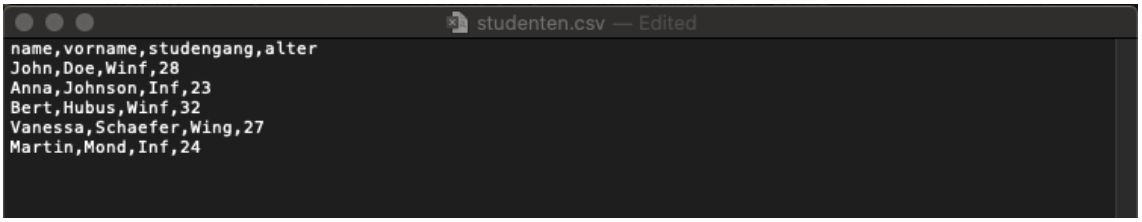

Abb. 4.5: Format einer CSV-Datei

Durch das Bestätigen der Schaltfläche *CSV Einlesen* erscheint ein Dateidialog, wie in Abbildung 4.6 zu sehen. Durch die Auswahl der entsprechenden Datei, wird diese eingelesen.

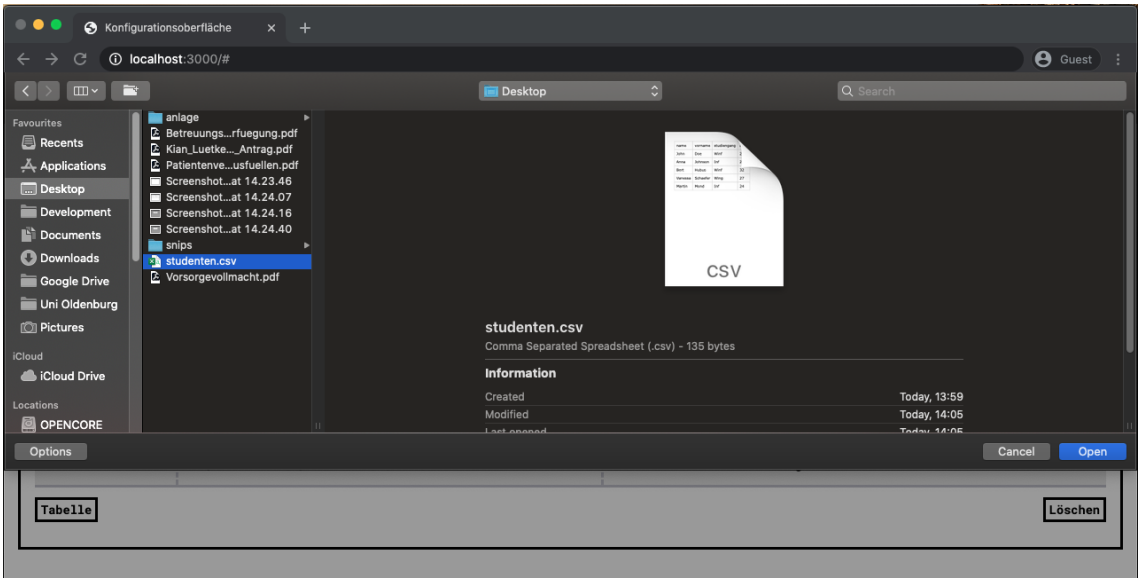

Abb. 4.6: Einlesen einer CSV-Datei

Nach erfolgreichem Hochladen der CSV-Datei erscheinen die Daten in einem neuen Modal. Hier besteht die Möglichkeit Änderungen vorzunehmen. Zudem muss der gewünschte Pfad (Name des Endpunktes) angegeben werden.

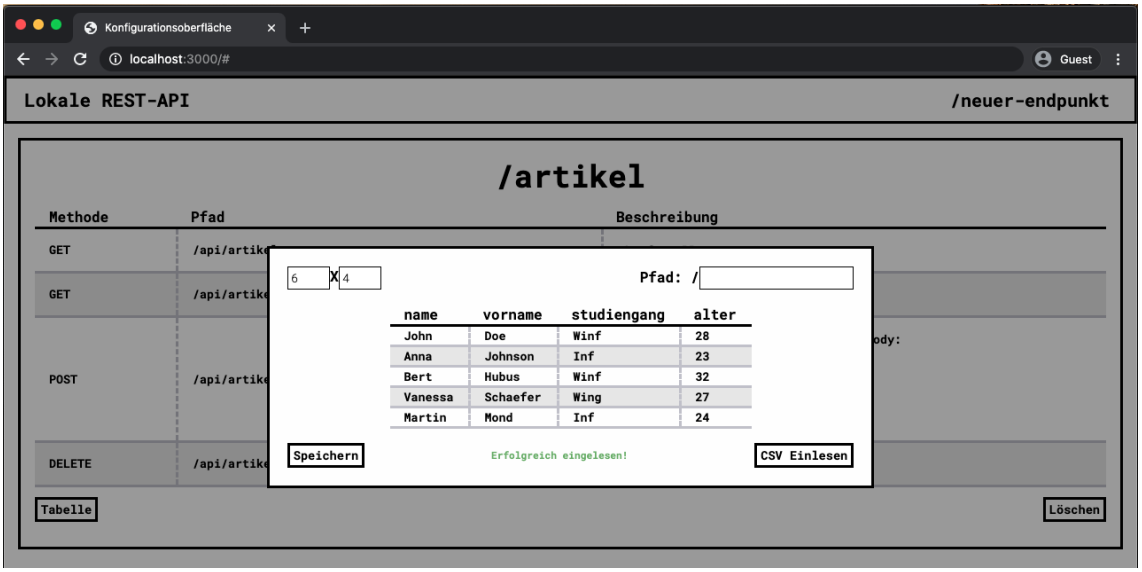

Abb. 4.7: Anzeige der importierten Tabelle

4.2 REST-API

4.2.1 Datenabfrage

Um die erstellten Endpunkte abzufragen, müssen HTTP-Requests auf die entsprechende URL ausgeführt werden. Die einfachste Form ist das Konsolen Programm *curl*. Dieses gehört bei den meisten Betriebssystemen zum Standard und ist auch unter dem hier verwendeten MacOS Teil des Systems. [Dan20]

```
> curl localhost:3000/api/artikel | json_pp
{
   "message" : "Request successful",
   "status" : 200,
   "data" : [
      {
         "STOCK" : "34",
         "NAME" : "Asus ACX5600 WiFi Router",
         "LAGER" : "433",
         "id" : 1
      },
      {
         "LAGER" : "8080",
         "NAME" : "TP-Link Archer AX6000",
         "STOCK" : "55",
         "id" : 2
      },
      [...]
   ]
}
```

Der obige Ausschnitt zeigt, wie ein einfacher GET-Request auf den Endpunkt *artikel*

ausgeführt wird. Durch die Übergabe an das ebenfalls standardmäßig vorhandene Konsolenprogramm *json_pp* wird die Ausgabe ansprechender formatiert. Dies ist nicht zwingend notwendig, verbessert jedoch die Übersichtlichkeit. Die Ausgabe zeigt eine Nachricht, dass die Anfrage fehlerfrei war, sowie den Status-Code 200. Zudem wird das Array *data* zurückgegeben, welches die angeforderten Daten enthält.
Um nun nur einen bestimmten Artikel auszugeben wird ein Querystring an den GET-Request angehängt:

```
> curl http://localhost:3000/api/artikel\?id\=1 |json_pp
{
   "status" : 200,
   "message" : "Request successful",
   "data" : [
      {
         "LAGER" : "433",
         "NAME" : "Asus ACX5600 WiFi Router",
         "id" : 1,
         "STOCK" : "34"
      }
   ]
}
```

Hervorzuheben ist, dass bei der Benutzung von *curl* Sonderzeichen, wie zum Beispiel das ? oder das & Zeichen ein vorgestelltes / Zeichen benötigen, damit die Abfrage korrekt funktioniert.

4.2.2 Datenmanipulation: Erstellen und Löschen

Um einen neuen Eintrag der Tabelle hinzuzufügen bedarf es eines POST-Requests. Dies beherrscht das Werkzeug *curl* ebenfalls. Folgendes Listing illustriert ein solches Vorgehen:

```
> curl -d '{"LAGER": "433", "NAME": "Huawei AX3 Standard", "STOCK": "4"}' \
-H "Content-Type: application/json" \
-X POST http://localhost:3000/api/artikel | json_pp
{
   "message" : "Ressource successfully created",
   "status" : 201,
   "data" : [
         [...]
         {
            "STOCK" : "4",
            "id" : 5,
            "LAGER" : "433",
            "NAME" : "Huawei AX3 Standard"
         }
      ]
}
```

Dabei werden einige Parameter genutzt:

- *-d*: das d steht für data, hier wird der JSON-Body übergeben.
- *-H*: An dieser stelle können HTTP-Header übergeben werden.
- *-X*: Dieser Buchstabe steht für den Request-Typ; in diesem Fall POST.

Wie in diesem Beispiel zu sehen, antwortet die API nicht nur mit einer Erfolgsmeldung, sondern gibt auch alle Daten zurück inklusive der neu hinzugefügten.
Schließlich fehlt noch das Löschen vorhandener Einträge. Dafür wird der HTTP-Request DELETE genutzt, wie folgendes Listing veranschaulicht:

```
> curl -X DELETE http://localhost:3000/api/artikel\?id\=2 | json_pp
{
   "status" : 200,
   "data" : [
      {
         "STOCK" : "21",
         "id" : 3,
         "LAGER" : "433",
         "NAME" : "Netgear Nighthawk AX8"
      },
      {
         "NAME" : "Huawei AX3 PRO",
         "STOCK" : "12",
         "LAGER" : "433",
         "id" : 4
      },
      {
         "NAME" : "Huawei AX3 Standard",
         "LAGER" : "433",
         "id" : 5,
         "STOCK" : "4"
      }
   ],
   "message" : "Request successful"
}
```

Hier lässt sich erkennen, dass wieder der -X Parameter genutzt wurde, um die Methode DELETE auszuwählen. Die API gibt eine Erfolgsmeldung zurück sowie die verbleibenden Daten der Tabelle.

5 Evaluation

5.1 Umsetzung

In diesem Abschnitt wird die Software auf die Einhaltung der in der Anforderungsanalyse festgelegten Eigenschaften geprüft.

Funktionale Anforderungen Die funktionalen Anforderungen beschreiben Eigenschaften, die mit der direkten Bedienung der Software einhergehen. In diesem Fall wurde festgelegt, dass die Software eine Oberfläche zur Konfiguration der Endpunkte bereitstellen soll. Unterstützt werden muss manuelles oder automatisiertes (durch einlesen einer CSV Datei) Hinzufügen von Endpunkten. [Som16] Zudem sind die Endpunkte visuell darzustellen und die Konfigurationen sollen persistent, über mehrere Programmstarts hinweg, erhalten bleiben.

Wie im Kapitel 4 - dem Handbuch - zu sehen, werden alle visuellen Punkte erfüllt. Sobald die Software startet stellt sie eine Oberfläche bereit, welche all diese Anforderungen abdeckt. Endpunkte können entweder manuell oder mittels CSV-Upload hinzugefügt werden. Die persistente Speicherung wurde durch das Nutzen einer *sqlite* Datenbank realisiert. Zusammenfassend lässt sich also feststellen, dass alle in der Anforderungsanalyse festgelegten funktionalen Anforderungen eingehalten worden sind.

Nicht-funktionale Anforderungen Diese Anforderungen stehen nicht direkt im Bezug zur Benutzung der Software, sondern treffen Aussagen über technische

Einschränkungen. Hier wurden drei Anforderungen gestellt. Das Ausführen der Software soll betriebssystemunabhängig und ohne bestehende Internetverbindung möglich sein. Des Weiteren sollen alle im Kapitel 2 festgelegten REST Standards eingehalten werden.

Auf die Anhaltung der REST Konventionen wurde im Kapitel 3 strikt geachtet. Da die Software einen lokalen Server startet, auf dem sowohl die Oberfläche als auch die API zur Verfügung gestellt werden, ist keine Internetverbindung erforderlich.
Der letzte Punkt - die Betriebssystemunabhängigkeit - hat einige Herrausforderungen geborgen, welche im nächsten Kapitel abschließend betrachtet werden.

5.2 Fazit

Dieser Abschnitt bewertet den verfolgten Ansatz und zeigt gegebenfalls Alternativen auf. Zunächst werden die genutzten Technologien untersucht. Um das Frontend und auch das Backend bereitzustellen kam in diesem Projekt das JavaScript Framework node.js zum Einsatz. Es hat sich als hervorragende Wahl herausgestellt, da alle Bestandteile innerhalb einer gemeinsamen Code-Basis gepflegt werden konnten. Durch die Einhaltung der MVC Architektur ist die Ordnerstruktur übersichtlich und modular aufgebaut.

Bei der Entwicklung des Frontends wurde auf modulares JavaScript und selbsterstelltes CSS gesetzt. Während sich die Funktionalität der Oberfläche sehr gut mit JavaScript umsetzen ließ, kostete die Erstellung eines eigenen CSS Stils sehr viel Zeit. Obwohl der BEM Ansatz zu guten konsistenten Ergebnissen geführt hat, wäre das Nutzen eines bereits vorhandenen CSS Frameworks wie z.B. *Bootstrap*[9] von Vorteil gewesen.
Die Wahl der Datenspeicherung ist in diesem Projekt auf die Nutzung einer sqlite Datenbank gefallen. Um mit node.js eine Anbindung an diese Datenbank

[9] https://getbootstrap.com/

sicherzustelllen wird ein Treiber benötigt, welcher in Abhängigkeit zum Betriebssystem kompiliert werden muss [Jos20].

Durch die überschaubare Anzahl der anfallenden Datenmengen, wäre eine einfache Datei im z.B. JSON Format sinnvoller gewesen. Somit hätte von einem Betriebssystem aus mit dem Werkzeug pkg ein ausführbares Programm für alle drei geforderten Betriebssysteme gleichzeitig erstellt werden können.
Dies wurde zwar durch die Erstellung mehrerer virtueller Maschinen umgangen; dieser Umstand hätte jedoch von Anfang an verhindert werden können.
Zusammenfassend lässt sich feststellen, dass sich die Software hat bewähren können und alle Anforderungen eingehalten hat. Durch die Ausarbeitung des Nutzerhandbuchs wird es potentiellen Usern leichtfallen Endpunkte zu erstellen und abzurufen. Kompilierte Versionen für die verschiedenen Betriebssysteme werden dieser Ausarbeitung beigefügt.

5.3 Ausblick

Dieses Projekt hat erfolgreich demonstriert, wie sich eine lokale REST-API entwicklen und benutzen lässt. Mit dieser Software ist es nun möglich Beispieldaten für eine Fremd-Anwendung bereitzustellen, welche vollständig modifizierbar sind.
Ein weiterführendes Projekt könnte diesen Ansatz vervollständigen und zum Beispiel Authentifizierungs-Methoden einführen, welche in der Oberfläche einstellbar wären. Denkbar wäre hier zum Beispiel eine *Digest Authentifizierung* [Net99].
Ein weiterer verfolgbarer Ansatz wäre die Erstellung einer Software oder mobilen App, welche die lokale REST-API nutzt, um Daten visuell darzustellen. Denkbar wäre zum Beispiel ein Online-Shop oder eine Monitoring-Applikation zur Überwachung von Industrieanlagen.

Literaturverzeichnis

[Dan20] Daniel Stenberg and Contributers. curl, 2020.

[Dog18] Fernando Doglio. *REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development*. Apress, Berkeley, CA, 2nd ed. edition, 2018.

[Fin17] Rich Finelli. *Mastering CSS*. Packt, Birmingham and Mumbai, October 2017.

[Ger06] Jason Gerner. *Professional LAMP: Linux, Apache, MySQL, and PHP Web development*. Programmer to programmer. Wiley, Indianapolis, Ind., 2006.

[Hun98] Ray Hunt. Internet/intranet firewall security—policy, architecture and transaction services. *Computer Communications*, 21(13):1107–1123, 1998.

[igo20] igorklopov. pkg - npm, 2020.

[Jos20] Wise Joschua. better-sqlite3 - npm, 2020.

[Mas12] Mark Massé. *REST API design rulebook: Designing consistent RESTful Web Service Interfaces*. O'Reilly, Beijing, 2012.

[Moz20] Mozilla MDN Webdocs. Javascript modules, 2020.

[Net99] Network Working Group. Rfc 2617 - http authentication: Basic and digest access authentication, 1999.

[Net05] Network Working Group. Rfc 3986 - uniform resource identifier (uri): Generic syntax, 2005.

[Net20a] Network Working Group. Rfc 2616 - hypertext transfer protocol – http/1.1, 27.3.2020.

[Net20b] Network Working Group. Rfc 4180 - common format and mime type for comma-separated values (csv) files, 27.3.2020.

[RBM16] John Resig, Bear Bibeault, and Josip Maras. *Secrets of the JavaScript ninja*. Manning Publications, Shelter Island, NY, second edition edition, 2016.

[Som16] Ian Sommerville. *Software engineering*. Always learning. Pearson, tenth edition, global edition edition, 2016.

[Vse20] Vsevolod Strukchinsky, Vladimir Starkov and contributors. Bem - block element modifier, 2020.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit wurde noch keinem anderen Prüfungsamt vorgelegt und auch nicht veröffentlicht.

Ort, Datum	Kian Lütke