manroland | GOSS

Projektdokumentation

Schulaufgabe Projekt REST Schnittstelle

Auftrag zur Umsetzung einer REST Schnittstelle für Kundenverwaltung

Fachinformatiker für Anwendungsentwicklung

Auszubildender
Pascal Masny
[redacted]

Ausbildungsbetrieb
manroland Goss web systems GmbH
Alois-Senefelder-Allee 1
86153 Augsburg

Abgabetermin
15.03.2023

Inhaltsverzeichnis

1 Einleitung 3
1.1 Projektumfeld 3
1.2 Projektziel 3
1.3 Projektbegründung 3
1.4 Projektschnittstellen 3
2 Projektplanung 4
2.1 Projektphasen 4
2.2 Ressourcenplanung 4
2.3 Planung des Backends 4
3 Projektanalyse 5
3.1 Soll-Zustand 5
4 Projektrealisierung 6
4.1 Initialisierung der Versionsverwaltung 6
4.2 Einrichten der Entwicklungsumgebung 6
4.3 Use Case-Diagramm 6
4.4 Sequenz-Diagramm 7
4.6 Softwarearchitektur 7
4.7 Geschäftslogik app.js 7
4.8 Geschäftslogik des Routers 8
4.9 Geschäftslogik des Datenbank Models 8
5 Projektabschluss 9
5.1 Validierung & Test 9
6 Anhang 10
6.1 Quellenverzeichnis 10
6.2 Ressourcenverzeichnis 10
6.3 Abbildungen 11
6.3.1 Kundenauftrag 11
6.3.2 Lastenheft 11
6.3.3 Code Struktur 12
6.3.4 app.js 12
6.3.5 orders.js 13
6.3.6 db.js 13
6.3.7 orders.model.js 14
6.3.8 dbconfig.js 14
6.3.9 Validierung der API 14

Abkürzungsverzeichnis

MGWS	manroland Goss web systems GmbH
BS-VII	Berufsschule-VII Augsburg
JSON	JavaScript Object Notation
REST	Representational State Transfer
API	Application Programming Interface
JS	JavaScript
NPM	Node Package Manager
URL	Uniform Resource Locators
DB	Daten Dank
SQL	Structured Query Language

1 Einleitung

1.1 Projektumfeld

Die Berufsschule BS-VII in Augsburg führt im Rahmen des IT-Projektfaches das Projekt "Schulaufgabe Projekt REST Schnittstelle" durch. Der Zeitraum für das Projekt ist vom 13.03.2023 bis zum 15.03.2023 geplant. Es stehen 10 Stunden zur Verfügung. Das Projekt umfasst Planung, Umsetzung, Projektierung, Dokumentation sowie ein Fachgespräch. Der Lehrer Herr Maik Aicher leitet das Projekt. Das Projekt ist erfunden und soll lediglich eine simuliert Kundenanfrage darstellen. Das Projekt dient als Ersatz für eine schriftliche Prüfung im Fach IT-Projekt und soll den Lernfortschritt messen. Das Ergebnis der Note fließt in die Endnote des Zeugnisses ein.

1.2 Projektziel

Ziel des Projektes ist das Erstellen einer REST-Schnittstelle zur Verwaltung von Kundendaten des Onlineshops WarpShop der Firma IT Solutons. Diese Schnittstelle soll bis zum 15.03.2023 realisiert werden.

Da die Kundenanfrage recht umfassend von den Forderungen ist, wurde sich entschieden, die einzelnen Unteraufgaben an die einzelnen Schüler der Klasse 12cAW zu verteilen. Als verantwortlicher Schüler für die Erstellung der API zur Übermittlung von OrderID's in einem vorgegebenen Bereich sollte man sich an die folgenden Anforderungen halten:

- HTTP-Statuscodes
- Infostrings bei Fehlermeldungen
- Begrenzung auf 100 Datensätze pro abfrage
- Logging für interne Entwicklungszwecke
- Trennung von Routen, Modell und Datenbank
- Übermittlung im JSON-Format

Hierzu soll im Backend ein ExpressJS Server neu ausgesetzt und konfiguriert werden. Anbei der in 1.3 Projektbegründung beschriebenen Kundenauftrag liegt auch ein Lastenheft (siehe 16.3.2 Lastenheft) bei.

1.3 Projektbegründung

Wie in 6.3.1 Kundenauftrag beschrieben, möchte die IT Solutions ihren WarpShop zukunftssicherer gestalten. Dafür hätte der Kunde gerne eine REST-Schnittstelle um mit der Datenbank zu kommunizieren. Dafür soll ein Prototyp entwickelt werden, die dann der IT Solutions zugeschickt werden soll.

1.4 Projektschnittstellen

Die technischen Rahmenbedingungen basieren auf dem ExpressJS Framework und der Programmiersprache JavaScript. Als Datenbank dient der lokal betriebener USB-Webserver der auf einer MySQL Datenbank basiert. Die Daten wurden im Voraus von der zur Verfügung gestellt und müssen nicht separat aggregiert werden. Die Daten, die von der REST-API aggregiert werden, werden mittels JSON und dem HTTP-Protokoll übermittelt.

Als Entwicklungsumgebung dient Visual Studio Code. Als Betriebssystem wird ein Firmenlaptop mit der neusten Version von Windows 10 Pro verwendet.

2 Projektplanung

2.1 Projektphasen

Für die Umsetzung des Projektes standen 10 Stunden zur Verfügung. Um diesen Zeitrahmen einzuhalten, wurde die Durchführung des Projektes in verschiedene Phasen gegliedert. Anhand dieser Phasen konnte der in Tabelle 1 dargestellte grober Zeitplan erstellt werden.

Projektphase	**Geplante Zeit in Stunden**
Analysephase	0.5
Entwurfsphase	0.5
Implementierungsphase	5
Testphase und Fehlerbehebung	2
Erstellung der Dokumentation	2
Gesamt	**10**

2.2 Ressourcenplanung

In der Übersicht, welche sich in 6.2. Ressourcenverzeichnis befindet, sind alle Ressourcen aufgelistet, die für das Projekt eingesetzt wurden.

Damit sind Hard- und Softwareressourcen gemeint. Bei der Auswahl der verwendeten Software wurde darauf geachtet, dass diese (falls möglich) kostenfrei (z. B. als Open Source) zur Verfügung stehen.

2.3 Planung des Backend

Für das Backend soll das ExpressJS Framework verwendet werde. Hierbei soll das http-Protokoll „GET“ verwendet, um die Daten aus der Datenbank zu holen. Dies wird über eine REST Schnittstelle realisiert. Der Befehl fürs Aggregieren der Daten soll so aussehen:
api/orders?idFr=myIdFrom&idTo=myIdTo

3 Projektanalyse

3.1 Soll-Zustand

Die zu entwickelnde REST API soll vor allem das Ziel, welches in 1.2 Projektziel angesprochen wurde, erfüllen. Da der Kunde IT Solutions schon eine vorhandene Datenbank in Ihrem System hat, soll diese verwendet werden. Diese Datenbank wird auf einen USB-Webserver basierend auf einer MySQL Datenbank bereitgestellt. Die Struktur der besagten Datenbank wird in Punkt 2.1 Soll-Zustand im 6.3.2 Lastenheft beschrieben.

Die Hauptnutzung der REST API soll die Verwaltung der Kundendaten der Mitarbeiter des Warpshops sein.

4 Projektrealisierung

4.1 Initialisierung der Versionsverwaltung

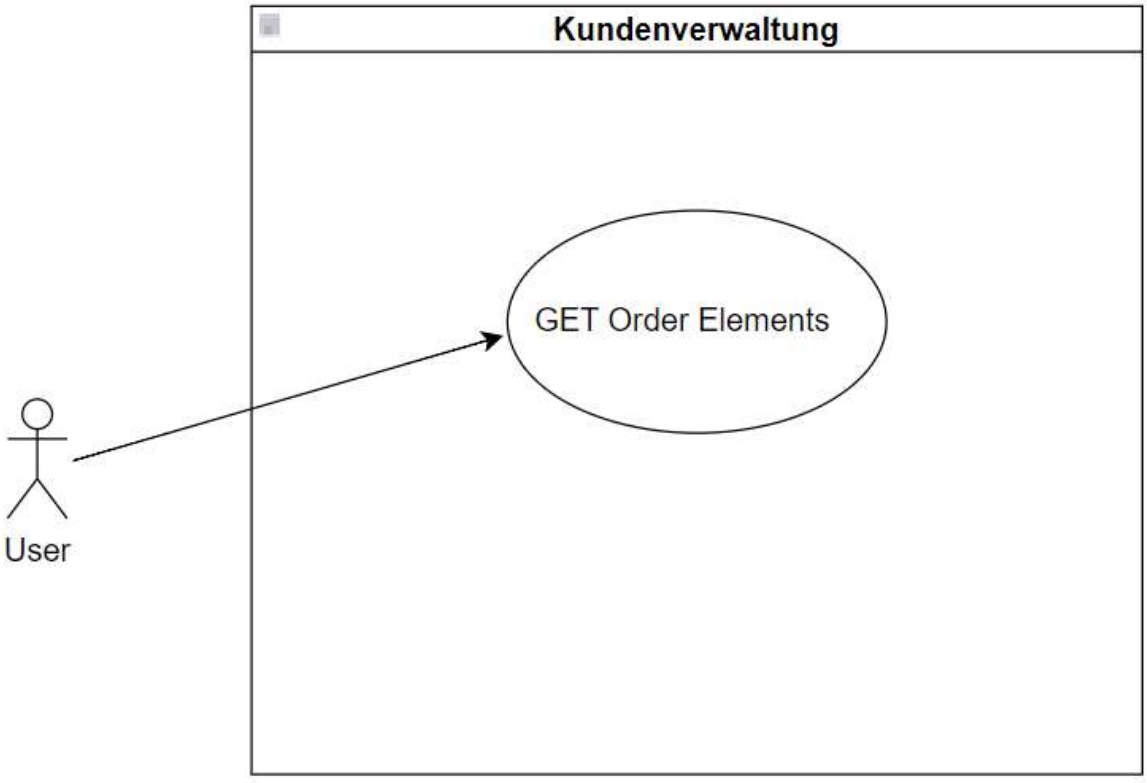
Das Versionsverwaltungssystem Git wird verwendet, um den aktuellen Stand des Projekts zu verfolgen. Zur Nutzung muss Git auf dem Computer installiert werden und das Verzeichnis des Projekts als Git-Repository initialisiert werden. Mit "git add" fügt man Dateien hinzu und "git commit" übernimmt Änderungen im Repository. Eine aussagekräftige Commit-Message ist empfehlenswert. Git ermöglicht eine effektive und systematische Verwaltung von Änderungen an dem Projekt.

4.2 Einrichten der Entwicklungsumgebung

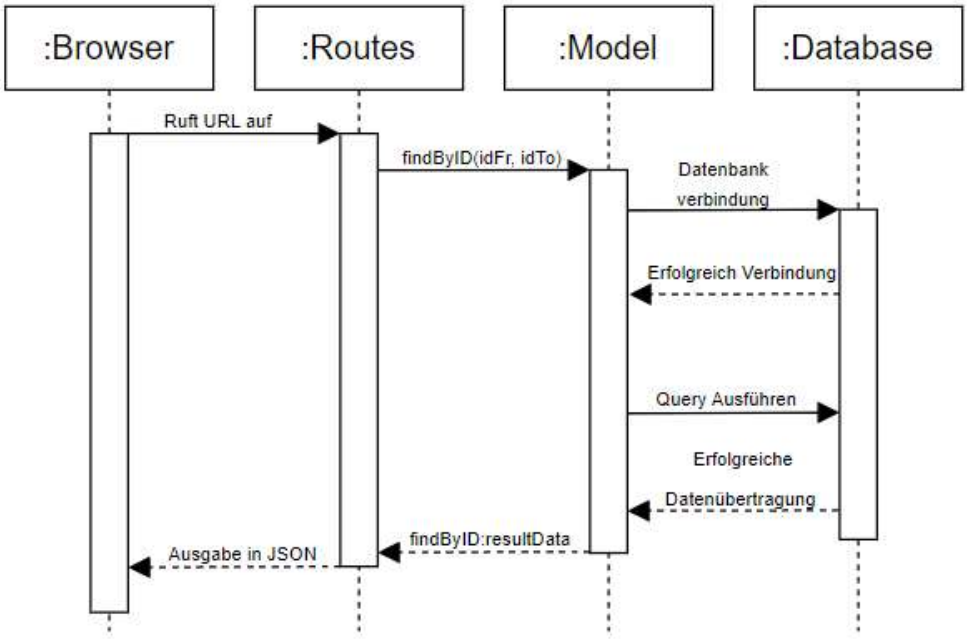
Die Entwicklungsumgebung befindet sich auf einem lokalen Windows-PC. Die für dieses Projekt benötigten Standardprogramme Visual Studio Code, JavaScript, NodeJS wurden über die Installer der Offiziellen Organisation Website installiert. Diese sind in 6.1 Quellenverzeichnis zu finden. Zudem wurden die in 6.2 Ressourcenverzeichnis beschriebenen JavaScript-Bibliotheken und Frameworks über das JavaScript-Paketverwaltungsprogramm **npm** installiert und sind global für alle JavaScript-Anwendungen für diesen PC nutzbar.

Die Einrichtung der Datenbank war nicht mehr nötig, da diese wie in 3.1 Soll-Zustand aufgezeigt schon bereitgestellt bekommen haben.

4.3 Use Case-Diagramm



4.4 Sequenz-Diagramm



4.5 Softwarearchitektur

Die Softwarearchitektur dieses Projekts ist in mehrere Teile aufgeteilt, um eine klare Trennung der Verantwortlichkeiten und eine bessere Wartbarkeit zu ermöglichen. Der Einstiegspunkt der Anwendung wird durch die Datei 6.3.4 app.js gebildet, die das Framework initialisiert und die Anwendung startet. Die HTTP REST-API-Anfragen werden von einem internen Router im Ordner **routes** verarbeitet und geleitet. Der Ordner **models** enthält die Datenbankmodelle sowie die SQL-Abfrage und den Datenbank-Connector, um eine Datenbankverbindung herzustellen. Der Ordner **config** enthält die Konfigurationen für die MySQL-Datenbank. Diese Aufteilung der Software in verschiedene Teile ermöglicht eine klarere Strukturierung und einfache Wartung des Codes. Diese Struktur kann in 6.3.3 Code Struktur gefunden werden.

4.6 Geschäftslogik app.js

Die Datei 6.3.4 app.js enthält den Quellcode für die Node.js-Anwendung, die eine RESTful-API bereitstellt, um OrderID's abzufragen. Zu Beginn des Codes wird das Express-Framework importiert und eine neue Express-App erstellt. Der Port, auf dem die Anwendung ausgeführt wird, ist auf **8081** festgelegt. Anschließend wird die Anwendung gestartet und auf eingehende Anforderungen auf dem definierten Port gelauscht. Wenn die Anwendung gestartet wird, wird eine Konsolenausgabe mit der Information ausgegeben, dass der Server auf dem angegebenen Port gestartet wurde. Schließlich wird der Router für die Bestellungen importiert und auf der URL **/api/orders** in der App registriert. Jede Anforderung, die an **/api/orders** gesendet wird, wird dann an den Router weitergeleitet, um die Geschäftslogik für die OrderID's zu verarbeiten.

4.7 Geschäftslogik des Routers

In der Datei 6.3.5 orders.js wird ein Express-Router erstellt, der für GET-Anfragen an den Pfad **/api/orders** zuständig ist. Das OrderElement-Modell wird importiert, um Bestellungen aus der Datenbank zu lesen. Die GET-Anfrage enthält zwei Query-Parameter **idFr** (idFrom) und **idTo** (idTo), um ID-Bereiche der Bestellungen abzufragen. Wenn die Query-Parameter fehlen oder ungültig sind, wird ein 400 Fehler zurückgegeben. Andernfalls werden sie in Integer-Werte umgewandelt und das OrderElement-Modell abgefragt, um Bestellungen mit ID-Werten zwischen **startId** und **endId** zu suchen. Bei Fehlern bei der Abfrage wird ein 500 Fehler zurückgegeben, sonst werden die Bestellungen als Antwort zurückgegeben. Der **orders-Router** wird auch als Middleware registriert, um jede eingehende Anforderung an den Pfad **/api/orders** an den Router weiterzuleiten, der die Anfrage verarbeitet und eine entsprechende Antwort sendet. Middleware ermöglicht die Wiederverwendung von Code und die Modularisierung von Funktionalitäten in separaten Modulen.

4.8 Geschäftslogik des Datenbank Models

Die Geschäftslogik dieses Projekts wird durch den Code in 6.3.4 db.js bereitgestellt. Diese Datei definiert eine allgemeine Datenbankverbindung, die in der gesamten Anwendung verwendet wird. Die Konfigurationsdaten für die Datenbankverbindung werden in einer separaten Datei namens 6.3.8 dbconfig.js definiert. Die Verbindung zur Datenbank wird hergestellt, sobald die Anwendung gestartet wird. Wenn ein Fehler auftritt, wird eine Fehlermeldung ausgegeben. Sobald die Verbindung erfolgreich hergestellt wurde, wird eine Erfolgsmeldung ausgegeben, um zu bestätigen, dass die Datenbankverbindung funktioniert. Die Geschäftslogik des Projekts ist eng mit der Datenbank verbunden, da die Datenbank die zentrale Komponente des Systems ist. Mit Hilfe des DB-Modells kann die Anwendung auf die Datenbank zugreifen. Es ist wichtig sicherzustellen, dass die Geschäftslogik und die Datenbank miteinander harmonieren, um sicherzustellen, dass die Anwendung reibungslos funktioniert und die Datenabfrage konsistent bleibt.

Das Model 6.3.4 orders.model.js ist Teil der Geschäftslogik der API, die implementiert werden muss. Dieses Modell definiert die Konstruktorenfunktion **OrderElement**, die verwendet wird, um Informationen zu Bestellelementen aus SQL-Ergebnissen zu extrahieren. Das Modell enthält auch eine statische Methode namens **findById**, die verwendet wird, um Bestellelemente basierend auf einer bestimmten Bestell-ID‘s zu suchen. Die Methode akzeptiert zwei Parameter: **id[]**, der den Bereich der Bestell-ID definiert, und **resultData**, das die Ergebnisse der SQL-Abfrage an den Aufrufer zurückgibt. Wenn die Methode ausgeführt wird, ruft sie die SQL-Abfrage ab, um die Bestellelemente abzurufen, die der angegebenen Bestell-ID‘s entsprechen. Wenn die Abfrage erfolgreich ist, werden die Ergebnisse an den Aufrufer zurückgegeben. Wenn jedoch ein Fehler auftritt, wird eine Fehlermeldung an den Aufrufer zurückgegeben. Das Modell enthält auch spezielle Bedingungen, um sicherzustellen, dass die Anzahl der zurückgegebenen Bestellelemente begrenzt wird, um die API-Leistung undSicherheit zu optimieren und Überlastung zu vermeiden. Wenn mehr als 100 Bestellelemente gefunden werden, gibt die Methode einen Fehler zurück.

5 Projektabschluss

5.1 Validierung & Test

Im Rahmen der Validierung und des Tests der REST API wurde ein White Box Test durchgeführt. Hierbei wurde die in der 2.3 Planung des Backends vorgegebene URL im Browser aufgerufen, um die Daten aus der Datenbank abzufragen. Der Befehl fürs Aggregieren der Daten wurde als "GET" http-Protokoll ausgeführt. Die erwarteten Werte wurden erfolgreich zurückgegeben.

Die zu abfragende URL lautete: **http://localhost:8081/api/orders?idFr=10293&idTo=15144**.
Der zu abfragende OrderID Bereich: **10293 - 15144**

Dabei wurden folgende aggregierten Daten:
[{"OrderID":10293,"ProductID":1004521,"Quantity":1},
{"OrderID":12678,"ProductID":1009390,"Quantity":1},
{"OrderID":12678,"ProductID":1005422,"Quantity":3},
{"OrderID":12678,"ProductID":1004293,"Quantity":1},
{"OrderID":12678,"ProductID":1002778,"Quantity":1},
{"OrderID":12678,"ProductID":1003418,"Quantity":1},
{"OrderID":12678,"ProductID":1003701,"Quantity":1},
{"OrderID":13565,"ProductID":1005839,"Quantity":2},
{"OrderID":15144,"ProductID":1008564,"Quantity":1},
{"OrderID":15144,"ProductID":1002593,"Quantity":1}].

Die Testergebnisse zeigen, dass die REST API wie geplant funktioniert und die erwarteten Daten erfolgreich abgerufen werden können. Diese Stimmen auch mit den Daten in der MySQL überein, wenn man diese händisch abfragen würden.
Siehe dazu 6.3.9 Validierung der API.

5.2 Abschluss

Das wurde erfolgreich innerhalb des vorgegebenen Zeitrahmens abgeschlossen und alle in 1.2 Projektziel beschriebenen Anforderungen wurden erreicht. Die Applikation funktioniert zuverlässig und erfüllt alle gestellten Anforderungen. Sowohl auf interner Server Seite sowohl auch über die REST-Schnittstelle.

Durch intensive Tests konnte nachgewiesen werden, dass die Ergebnisse korrekt und repräsentativ sind. Dadurch wird sichergestellt, dass die API auch in der Praxis einwandfrei und fehlerfrei arbeitet. Zusammenfassend kann festgehalten werden, dass das Projekt ein voller Erfolg war und alle gestellten Ziele erreicht wurden.

6 Anhang

6.1 Quellenverzeichnis

Name	Link
JavaScript	https://js.org/
NodeJS	https://nodejs.org/en/
npm	https://www.npmjs.com/
JSON	https://www.json.org/json-de.html
ExpressJS	https://expressjs.com/de/
GitHub	https://github.com/
StackOverflow	https://stackoverflow.com/
Microsoft Windows	https://www.microsoft.com/de-de/
Arbeitsblätter der BS-VII	

6.2 Ressourcenverzeichnis

Hardware:

- Arbeitsplatz mit Internetzugang in der Berufsschule VII
- Fujitsu Laptop (Lifebook E Sereies) der MGWS
- Tuxedo Laptop (Privat)
- Apple iPad (Privat)

Software:

- Betriebssystem: Windows 10 Pro
- Entwicklungsumgebung: Visual Studio Code
- Dokumentation und Präsentation: Microsoft Office Professional Plus 2019
- Programmiersprache + Interpreter: JavaScript + NodeJS
- Paket & Bibliotheksmanager: npm

Bibliotheken:

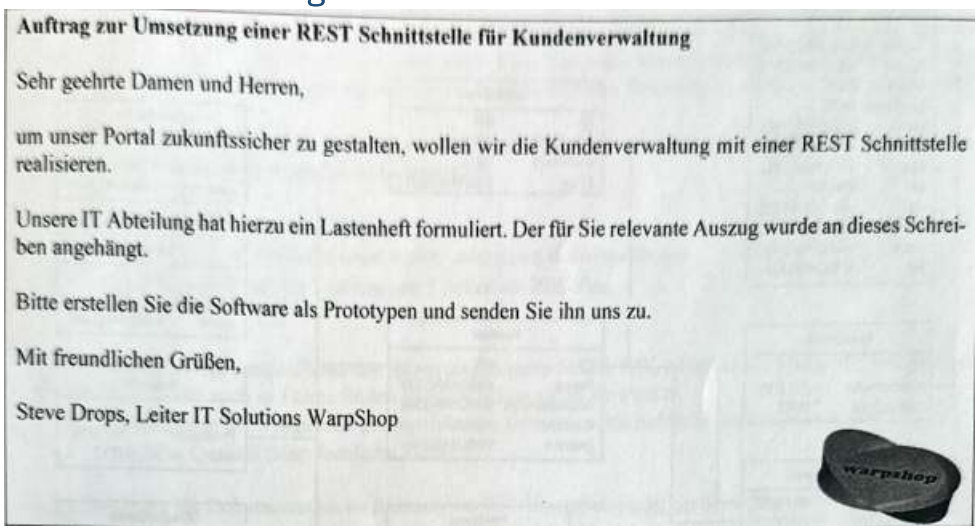
- Framework: ExpressJS
- Datenabfrage: JSON
- Datenbank: USB-Webserver (MySQL)

Version Control & Management:

- GitHub (Git)

6.3 Abbildungen

6.3.1 Kundenauftrag



Auftrag zur Umsetzung einer REST Schnittstelle für Kundenverwaltung

Sehr geehrte Damen und Herren,

um unser Portal zukunftssicher zu gestalten, wollen wir die Kundenverwaltung mit einer REST Schnittstelle realisieren.

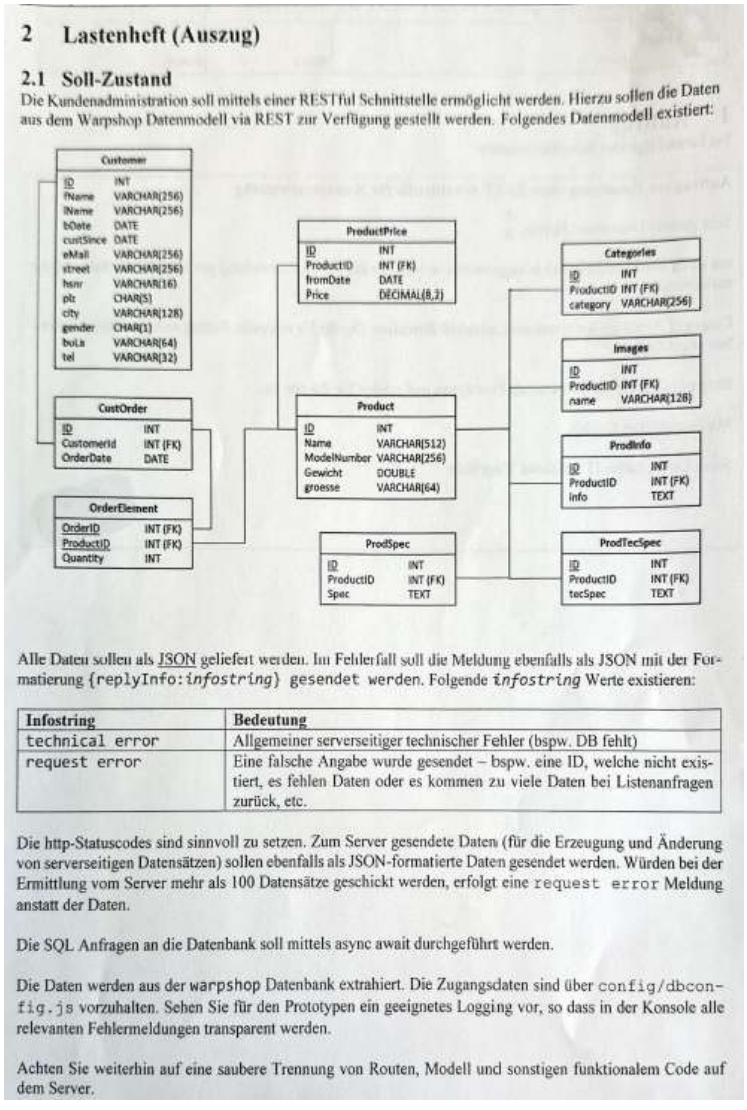
Unsere IT Abteilung hat hierzu ein Lastenheft formuliert. Der für Sie relevante Auszug wurde an dieses Schreiben angehängt.

Bitte erstellen Sie die Software als Prototypen und senden Sie ihn uns zu.

Mit freundlichen Grüßen,

Steve Drops, Leiter IT Solutions WarpShop

6.3.2 Lastenheft



2 Lastenheft (Auszug)

2.1 Soll-Zustand

Die Kundenadministration soll mittels einer RESTful Schnittstelle ermöglicht werden. Hierzu sollen die Daten aus dem Warpshop Datenmodell via REST zur Verfügung gestellt werden. Folgendes Datenmodell existiert:

Alle Daten sollen als JSON geliefert werden. Im Fehlerfall soll die Meldung ebenfalls als JSON mit der Formatierung `{replyInfo:infostring}` gesendet werden. Folgende `infostring` Werte existieren:

Infostring	Bedeutung
`technical error`	Allgemeiner serverseitiger technischer Fehler (bspw. DB fehlt)
`request error`	Eine falsche Angabe wurde gesendet – bspw. eine ID, welche nicht existiert, es fehlen Daten oder es kommen zu viele Daten bei Listenanfragen zurück, etc.

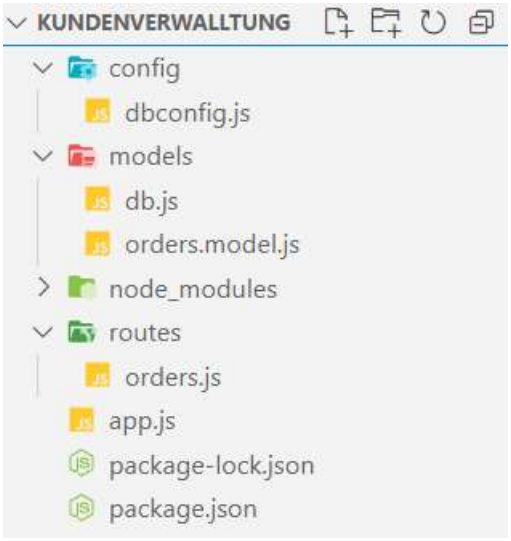
Die http-Statuscodes sind sinnvoll zu setzen. Zum Server gesendete Daten (für die Erzeugung und Änderung von serverseitigen Datensätzen) sollen ebenfalls als JSON-formatierte Daten gesendet werden. Würden bei der Ermittlung vom Server mehr als 100 Datensätze geschickt werden, erfolgt eine `request error` Meldung anstatt der Daten.

Die SQL Anfragen an die Datenbank soll mittels async await durchgeführt werden.

Die Daten werden aus der `warpshop` Datenbank extrahiert. Die Zugangsdaten sind über `config/dbconfig.js` vorzuhalten. Sehen Sie für den Prototypen ein geeignetes Logging vor, so dass in der Konsole alle relevanten Fehlermeldungen transparent werden.

Achten Sie weiterhin auf eine saubere Trennung von Routen, Modell und sonstigen funktionalem Code auf dem Server.

6.3.3 Code Struktur



6.3.4 app.js

app.js > ...

```
1   const express = require('express');
2   const app = express();
3   const EXP_PORT = 8081;
4   const orders = require("./routes/orders.js");
5
6   // Starting the server and listening for incoming requests on the defined port number
7   app.listen(EXP_PORT, () => {
8       console.log("Server Port: " + EXP_PORT);
9   })
10
11  // Mounting the 'orders' router on the '/api/orders' route of the application.
12  app.use("/api/orders", orders);
```

6.3.5 orders.js

routes > orders.js > ...

```
const express = require('express');
const router = express.Router();
const OrderElement = require("../models/orders.model.js");


// Handling GET requests made to the '/api/orders' route of the application.
router.get("/", (req, res) => {
  // Extracting 'idFr' and 'idTo' query parameters from the request object.
  const { idFr, idTo } = req.query;

  // Checking if 'idFr' and 'idTo' query parameters are valid integers.
  if (isNaN(idFr) || isNaN(idTo)) {
    res.status(400).send("request error");
    return;
  }

  // Parsing the start and end id values to integers.
  const startId = parseInt(idFr);
  const endId = parseInt(idTo);

  // Querying the 'OrderElement' model to find all elements whose 'id' values are between the start and end id values.
  OrderElement.findById({ start: startId, end: endId }, (err, orderElements) => {
    if (err) {
      // If there is an error, sending a 500 status code and the error message.
      console.log("DB Error: " + err);
      res.status(500).send(err);
    } else {
      // If there are no errors, sending the retrieved order elements as a response.
      res.status(200).send(orderElements);
      console.log("Good resquest with: " + idFr + " & ", idTo);
    }

  });

});
```

6.3.6 db.js

models > db.js > ...

```
const mysql = require("mysql");
const dbConfig = require("../config/dbconfig.js");

const connection = mysql.createConnection({
host: dbConfig.HOST,
port: dbConfig.PORT,
user: dbConfig.USER,
password: dbConfig.PASSWORD,
database: dbConfig.DB
});

connection.connect(error => {
    if (error) throw error;
    console.log("DB conn works");
});

module.exports = connection;
```

6.3.7 orders.model.js

models > orders.model.js > ...

```
// Importing the 'db.js' module that establishes connection with the database.
const dbcon = require("./db.js");

// Defining the OrderElement constructor function that takes SQL results and extracts relevant information.
const OrderElement = function(OrderElementData) {
    this.OrderID = OrderElementData.OrderID;
    this.ProductID = OrderElementData.ProductID;
    this.Quantity = OrderElementData.Quantity;
};

// Adding a static 'findById' method to the OrderElement constructor function.
OrderElement.findById = async (id, resultData) => {
    // Defining the SQL query to retrieve order elements whose 'OrderID' values are between 'id.start' and 'id.end'.
    const query = 'SELECT * FROM orderelement WHERE OrderID >= ? AND OrderID <= ? ORDER BY OrderID ASC';
    const params = [id.start, id.end];

    // Executing the SQL query using the 'dbcon.query' method and passing in the 'id' and 'resultData' parameters.
    dbcon.query(query, params, (err, result) => {
        // If there is an error, logging it to the console and returning the error message to the 'resultData' callback function.
        if (err) {
            resultData({ replyinfo: replyInfo.technical_error, test: "technical error" }, null);
            console.log(err);
            return;
        }

        // If there are no order elements found, returning a request error to the 'resultData' callback function.
        if (result.length === 0) {
            resultData({ replyinfo: replyInfo.request_error }, null);
            console.log("request error");
            return;
        }

        // If there are more then 100 results elements, throw an error
        if (result.length > 100) {
          resultData({ replyinfo: replyInfo.request_error, message: "request error" }, null);
          console.log("Too many results");
          return;
       }

        // If order elements are found, mapping over each element and creating a new OrderElement object for each one.
        const orderElements = result.map((orderElementData) => new OrderElement(orderElementData));
        // Returning the order elements as an array to the 'resultData' callback function.
        resultData(null, orderElements);
        return;
    });
};

// Exporting the OrderElement constructor function as a module.
module.exports = OrderElement;
```

6.3.8 dbconfig.js

config > dbconfig.js > ...

```
module.exports = {
    HOST: "localhost",
    PORT: "3306",
    USER: "root",
    PASSWORD: "usbw",
    DB: "warpshop_red"
};
```

6.3.9 Validierung der API