### Quellcode

| **Feld** | **Beschreibung** |
| --- | --- |
| **Name des Artefakts** | Quellcode |
| **Verantwortlich** | Software-Entwickler |
| **Ziel des Dokuments** | Entwicklung des Programmcodes zur Umsetzung der Anforderungen und Spezifikationen. |
| **Wesentliche Inhalte** | - Programmcode  - Dokumentation des Codes  - Kommentare und Anmerkungen im Code |
| **Erstellungszeitpunkt** | Entwicklungsphase des Projekts, nach Erstellung der technischen Spezifikationen |
| **Nutzung im Projektverlauf** | Grundlage für die technische Implementierung und die spätere Wartung und Weiterentwicklung der Anwendung |
| **Abnahme** | Durch die IT-Systemverantwortlichen und die Entwickler; Überprüfung der Vollständigkeit und Korrektheit des Codes |
| **Abhängigkeiten** | - Technische Spezifikationen  - API-Spezifikationen |
| **Versionierung** | Versionskontrolle mittels Versionsmanagement-Tools (z.B. Git) |
| **Vorlagen und Tools** | Entwicklungsumgebungen (IDEs) und Versionskontrollsysteme (z.B. Git, GitHub) |
| **Zusätzliche Anmerkungen** | Regelmäßige Code-Reviews und Tests notwendig, um die Qualität und Sicherheit des Codes zu gewährleisten |

### Verzeichnisstruktur

lua

crm-scoring-system/

├── config/

│ └── config.js

├── controllers/

│ └── scoringController.js

├── models/

│ ├── customer.js

│ ├── errorLog.js

│ └── index.js

├── routes/

│ └── scoringRoutes.js

├── utils/

│ └── dataValidator.js

├── uploads/

├── app.js

├── package.json

└── README.md

### 2. Initiale Konfigurationsdatei

config/config.js

js

module.exports = {

development: {

username: 'root',

password: 'password',

database: 'crm\_db',

host: '127.0.0.1',

dialect: 'mysql'

},

production: {

username: 'root',

password: 'password',

database: 'crm\_db',

host: '127.0.0.1',

dialect: 'mysql'

}

};

### 3. Datenbankmodelle

models/customer.js

js

const { Sequelize, DataTypes } = require('sequelize');

const sequelize = require('./index').sequelize;

const Customer = sequelize.define('Customer', {

id: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true

},

name: {

type: DataTypes.STRING,

allowNull: false

},

address: {

type: DataTypes.STRING,

allowNull: true

},

scoringValue: {

type: DataTypes.DECIMAL(5, 2),

allowNull: true

}

}, {

tableName: 'customers',

timestamps: false

});

module.exports = Customer;

models/errorLog.js

js

const { Sequelize, DataTypes } = require('sequelize');

const sequelize = require('./index').sequelize;

const ErrorLog = sequelize.define('ErrorLog', {

id: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true

},

customerId: {

type: DataTypes.INTEGER,

allowNull: false

},

errorType: {

type: DataTypes.STRING,

allowNull: false

},

date: {

type: DataTypes.DATE,

allowNull: false,

defaultValue: Sequelize.NOW

}

}, {

tableName: 'error\_logs',

timestamps: false

});

module.exports = ErrorLog;

models/index.js

js

const { Sequelize } = require('sequelize');

const config = require('../config/config')[process.env.NODE\_ENV || 'development'];

const sequelize = new Sequelize(config.database, config.username, config.password, {

host: config.host,

dialect: config.dialect

});

const Customer = require('./customer');

const ErrorLog = require('./errorLog');

module.exports = {

sequelize,

Customer,

ErrorLog

};

### 4. Datenvalidierung

utils/dataValidator.js

js

const validateScoringData = (data) => {

const errors = [];

data.forEach((row, index) => {

if (!row.kundenId) {

errors.push({ row: index + 1, column: 'kundenId', error: 'Kunden\_ID not found.' });

}

// Additional validation logic can be added here

});

return errors;

};

module.exports = {

validateScoringData

};

### 5. Controller

controllers/scoringController.js

js

const { Customer, ErrorLog } = require('../models');

const { validateScoringData } = require('../utils/dataValidator');

const readXlsxFile = require('read-excel-file/node');

const uploadScoringData = async (req, res) => {

try {

const rows = await readXlsxFile(req.file.path);

const data = rows.slice(1).map(row => ({

kundenId: row[0],

scoringValue: row[1]

}));

const errors = validateScoringData(data);

if (errors.length > 0) {

return res.status(400).json({ status: 'error', message: 'File validation failed.', errors });

}

for (const row of data) {

const customer = await Customer.findByPk(row.kundenId);

if (customer) {

customer.scoringValue = row.scoringValue;

await customer.save();

} else {

await ErrorLog.create({

customerId: row.kundenId,

errorType: 'Customer not found'

});

}

}

res.status(200).json({ status: 'success', message: 'File uploaded and data validated successfully.' });

} catch (error) {

res.status(500).json({ status: 'error', message: 'Internal server error.' });

}

};

const getCustomerScore = async (req, res) => {

try {

const customer = await Customer.findByPk(req.params.id);

if (!customer) {

return res.status(404).json({ status: 'error', message: 'Customer not found.' });

}

res.status(200).json({

customerId: customer.id,

scoringValue: customer.scoringValue,

lastUpdated: customer.updatedAt

});

} catch (error) {

res.status(500).json({ status: 'error', message: 'Internal server error.' });

}

};

const updateCustomerScore = async (req, res) => {

try {

const customer = await Customer.findByPk(req.params.id);

if (!customer) {

return res.status(404).json({ status: 'error', message: 'Customer not found.' });

}

customer.scoringValue = req.body.scoringValue;

await customer.save();

res.status(200

js

).json({ status: 'success', message: 'Scoring value updated successfully.' });

} catch (error) {

res.status(500).json({ status: 'error', message: 'Internal server error.' });

}

};

module.exports = {

uploadScoringData,

getCustomerScore,

updateCustomerScore

};

### 6. Routen

routes/scoringRoutes.js

js

const express = require('express');

const multer = require('multer');

const path = require('path');

const { uploadScoringData, getCustomerScore, updateCustomerScore } = require('../controllers/scoringController');

const router = express.Router();

const storage = multer.diskStorage({

destination: (req, file, cb) => {

cb(null, 'uploads/');

},

filename: (req, file, cb) => {

cb(null, `${Date.now()}-${path.extname(file.originalname)}`);

}

});

const upload = multer({ storage });

router.post('/upload', upload.single('file'), uploadScoringData);

router.get('/customers/:id/score', getCustomerScore);

router.put('/customers/:id/score', updateCustomerScore);

module.exports = router;

### 7. Hauptanwendung

app.js

js

const express = require('express');

const bodyParser = require('body-parser');

const { sequelize } = require('./models');

const scoringRoutes = require('./routes/scoringRoutes');

const app = express();

const port = process.env.PORT || 3000;

app.use(bodyParser.json());

app.use('/api', scoringRoutes);

// Synchronize models with the database

sequelize.sync().then(() => {

console.log('Database synchronized');

});

app.listen(port, () => {

console.log(`Server running on http://localhost:${port}`);

});

### 8. Paketkonfigurationsdatei

package.json

json

{

"name": "crm-scoring-system",

"version": "1.0.0",

"description": "Erweiterung des CRM-Systems um die Befüllung mit Scoring-Daten",

"main": "app.js",

"scripts": {

"start": "node app.js"

},

"dependencies": {

"body-parser": "^1.19.0",

"express": "^4.17.1",

"multer": "^1.4.3",

"mysql2": "^2.1.0",

"read-excel-file": "^5.2.8",

"sequelize": "^6.6.2"

},

"author": "Max Mustermann",

"license": "ISC"

}

### 9. Readme-Datei

README.md

markdown

# CRM Scoring System

## Überblick

Dieses Projekt erweitert ein CRM-System um die Möglichkeit, Scoring-Daten von Kunden über eine Excel-Datei hochzuladen und zu speichern.

## Installation

1. Repository klonen:

```bash

git clone <repository-url>

cd crm-scoring-system

1. Abhängigkeiten installieren:

bash

 npm install

 Datenbank konfigurieren:

* Bearbeiten Sie die config/config.js Datei und fügen Sie Ihre Datenbankdetails hinzu.

 Server starten:

bash

1. npm start

## API-Endpunkte

### POST /upload

* Beschreibung: Hochladen einer Excel-Datei mit Scoring-Daten.
* Header: Authorization: Bearer <token>
* Body: Multipart-Form-Data mit der Datei.

### GET /customers/:id/score

* Beschreibung: Abrufen des Scoring-Werts eines bestimmten Kunden.
* Header: Authorization: Bearer <token>

### PUT /customers/:id/score

* Beschreibung: Manuelles Aktualisieren des Scoring-Werts eines bestimmten Kunden.
* Header: Authorization: Bearer <token>
* Body: JSON mit dem neuen Scoring-Wert.

## Entwickler

* **Max Mustermann** - Initiale Arbeit - [MaxMustermann](https://github.com/MaxMustermann)