1 Vorbereitungen

Für unsere MySQL Verbindung benötigen wir erstmal eine einfache Test-Datenbank. Diese finden Sie unter `CreateTestDb.sql`. Weiterhin müssen wir den MySQL Treiber für node.js installieren:

```
PS C:\temp\myServer> npm install mysql
```

Als nächstes erzeugen wir einen Unterordner für die Konfigurationsinformationen „`config`“ und legen dort das File `dbconfig.js` ab:

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

Dieses File exportiert also ein Objekt, welches die Verbindungsdaten zur Datenbank vorhält. Nachdem wir dieses File nicht in GIT hochladen wollen, fügen wir im Rootverzeichnis des Proejektes ein `.gitignore` File an, welches lediglich den Eintrag `config/dbconfig.js` beinhaltet (und für das Ausklammern der Bibliothekenverzeichnisses idealerweise noch `node_modules`).

2 Einbinden von mysql in die Applikation

Um die Datenbankverbindung sinnvoll zu kapseln, erzeugen wir einen Ordner „`models`“, welches die Repräsentation unserer Businessdaten darstellt. Hier fügen wir das File `db.js` ein, welches die Datenbankverbindung erzeugt:

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

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

// Datenbankverbindung öffnen
connection.connect(error => {
    if (error) throw error;
    console.log("Successfully connected to the database.");
  });

// Export der offenen Datenbanverbindung
module.exports = connection;
```

Danach ist es meist sinnvoll, für unsere Daten ein eigenes Objektmodell zu erstellen. Hierzu erzeugen wir ein File `musician.model.js` und hinterlegen dort neben dem passenden js-Objekt auch die notwendigen Zugriffsmöglichkeiten über die Datenbank:

```
const dbcon = require("./db.js");

// Konstruktor, basierend auf den SQL-Ergebnissen
const Musician = function(musicianData) {
  this.musId = musicianData.musId;
  this.firstName = musicianData.firstName;
  this.lastName = musicianData.lastName;
};

// Statische Methode. Sie erwartet eine id, nach der in der DB gesucht
// wird. Weiterhin eine Funktion „resultData“, welche die Ergebnisse
// verarbeiten soll
Musician.findById = (id, resultData) => {
  // DB-Abfrage über Parameter, um SQL injection zu verhindern[1]
  dbcon.query('SELECT FName AS firstName, LName AS lastName
  FROM musician WHERE id = ?', id, (err, result) => {
    if (err) {  // für den Fall, dass die Abfrage fehlschlug
      // Einfügen der Fehlermedlung in die Verarbeitungsfunktion
      resultData(err, null);
      return; // Abbruch von findById
    }

    if (result.length) {  // für den Erfolgsfall
      // Erzeugen eines neuen Objektes mit den Ergebnissen[2]
      resultData(null, new Musician({musId: id, ...result[0]}));
      return;
    }
    // für den Fall, dass nichts gefunden wurde
    resultData({ error_reason: "not_found" }, null);
  });
};

// Beispielmethode für den Fall, dass mehrere Daten zu erwarten sind
Musician.findByFirstName = (firstName, resultData) => {
  dbcon.query('SELECT ID AS musId, LName AS lastName
  FROM musician WHERE FName = ?', firstName, (err, result) => {
    if (err) {
      resultData(err, null);
      return;
    }
    // die Daten werden in einem Array gepusht und zurückgegeben
    let musicians = Array();
    if (result.length) {
      result.forEach((element) => {
        musicians.push(new Musician({firstName: firstName, ...element}));
      });
      resultData(null, musicians);
      return;
    }

    resultData({ error_reason: "not_found" }, null);
  });
};
```

[1] „named placeholders” werden im mysql Treiber (Version 2.18.1) nicht unterstützt, kann aber nachträglich ergänzt werden: https://coderwall.com/p/196wpg/enable-named-placeholders-in-node-mysql

[2] Der … Operator wird weiter unten erklärt.

```
module.exports = Musician;
```

Hier noch einige Hinweise zum Code. Das Model besteht aus einem Datenobjekt „`Musician`“, welches über einen Konstruktor erzeugt wird. Dieser hat „nur“ einen Parameter, welcher wiederum ein Objekt ist. Die Daten aus dem `SELECT` Statement erhalten wir über die `.query()` Methode ebenfalls als benamstes Objekt, deren Namen aus dem `SELECT` Aliasnamen der Spalten ermittelt werden. Insofern sorgen wir dafür, dass alle Ergebnisspalten aus der Datenbank-Query einen eindeutigen Namen haben.
Bei der Erzeugung des Objektes, können wir nun den Spread Operator „`...`“ verwenden. Dieser funktioniert wie folgt, wenn wir bspw. den Aufruf `{musId: id, ...result[0]}` durchführen:

- Erzeuge ein Objekt, da der Gesamtausdruck in geschweiften Klammern steht.
- Übernehme alle Eigenschaften aus dem (assoziativen) Array vor dem …Operator (in unserem Fall das Objekt in der ersten Position des result-Arrays, also `result[0]`).
- Übernehme alle Eigenschaften, welche explizit angegeben wurden (in unserem Fall wird `musId` mit dem Wert aus `id` belegt) und überschreibe ggf. bereits existierende Eigenschaften – es wird also in das übergebene assoziative Array ein neuer Eintrag mit `musId: id` ergänzt.

Weiterhin wurde die Verarbeitung der Ergebnisse über einen Funktionsparameter realisiert. Der Code

```
Musician.findById = (id, resultData) => {
```

besagt, dass die Funktion „`findById`“ zwei Parameter hat, `id` und `resultData`. Letzteres wird innerhalb des Codes aber als Funktion genutzt:

```
resultData(err, null);
```

Das bedeutet, dass beim Aufruf von `findById()` der zweite Parameter eine Funktion sein muss. Sehen wir uns hierzu den Aufruf an. Da wir den Code übersichtlich und modular gestalten wollen, schreiben wir den Verarbeitungscode für die `Musician` Objekte in einer eigenen Route-Datei unter dem Unterordner „`routes`“. Hier die Datei `musician.js`:

```
const express = require('express')
const router = express.Router();
const Musician = require("../models/musician.model");

// Für Routen, welche Musiker nach ID suchen:
router.get("/id/:id", (req, res, next) => {
  const {id} = req.params;
  // Aufruf der Modelmethode, welche einen Musiker über ID sucht
  Musician.findById(id, (err, musician) => {
    if (err) {
      console.log("DB Error: " + err);
      res.send("Technical Error");
    } else {
      res.send(musician.firstName + " " + musician.lastName);
    }
  });
});

// für Routen, welche Musiker nach Vornamen suchen:
router.get("/firstname/:firstName", (req, res, next) => {
  const {firstName} = req.params;
  Musician.findByFirstName(firstName, (err, musicians) => {
    if (err) {
      console.log("DB Error: " + err);
      res.send("Technical Error");
    } else {
      // String für die Ausgabe. In realer App hier rendern!
      let data = "";
      musicians.forEach((element) => {
```

```
        data += element.musId + " " + element.firstName + " " +
        element.lastName + "<br/>";
      });
      res.send(data);
    }
  });
});

module.exports = router;
```

Hier finden wir den Aufruf mit einer Funktion als Parameter:

```
  Musician.findById(id, (err, musician) => {
```

Zwischen den geschweiften Klammern ist die Funktion, welche im Model als `resultData` Funktion aufgerufen wird.

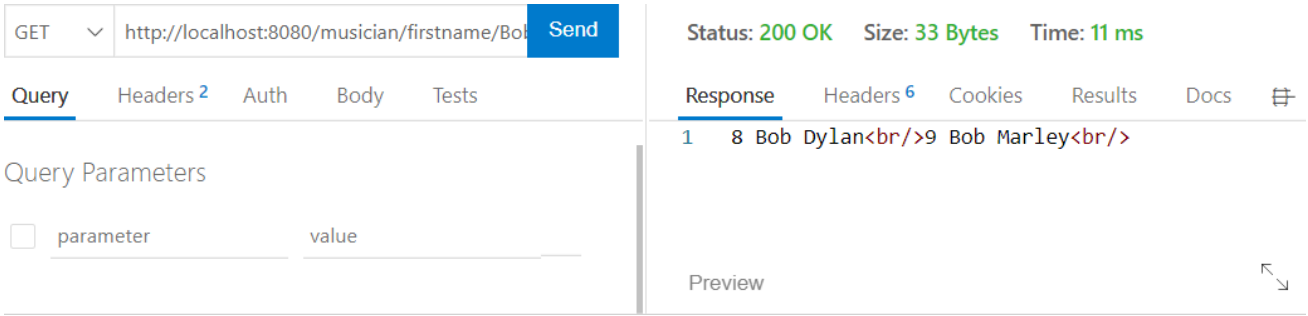
Zum Schluss fehlt nun noch die app.js, welche unsere Route als Middleware einbindet:

```
const express = require('express');
const app = express();
const EXP_PORT = 8080;

const musicianRoutes = require("./routes/musician");
app.use("/musician/", musicianRoutes);

app.listen(EXP_PORT, () => {
    console.log("Ich höre auf Port! " + EXP_PORT);
});
```

Hinweis: In großen Applikationen sollten die Datenbankverbindungen über einen Pool vorgehalten werden. Für unsere Applikation ist dies nicht notwendig, Details finden sich online, bspw. unter https://www.npmjs.com/package/mysql#pooling-connections

3 Test der Applikation

Da wir reine GET-Requests durchführen, können wir die relevanten URLs einfach in ein Browserfenster eingeben. Eine weitere Möglichkeit bieten uns auch http-Clients wie Postman oder das VSCode Plugin „Thunderclient“. Letzeres können wir einfach in VSCode installieren und über das User-Interface die URL eingeben. Diese Clients zeigen uns dann 1:1 den gesendeten Datenstrom an:



Der Vorteil eines derartigen http-Clients ist, dass wir sehr einfach POST-Anfragen zusammenstellen können, ohne dafür die notwendige HTML-Form zu realisieren. Für die serverseitige Entwicklung ist das oftmals der schnellste Weg, mögliche Fehler zu finden.