Node.js: Datenbankzugriff auf MySQL

- Voraussetzungen verstehen damit auf MySQL-Datenbank über Node.js zugegriffen werden kann
- Wissen wie der Zugriff mit sofortiger Reaktion auf die Antwort des DBMS erfolgen muss
- Wissen wie der Zugriff auf das DBMS unter Berücksichtigung des MVC-Patterns erfolgen muss
- Den Zugriff auf das DBMS über Promises realisieren können
- Mit der Utility-Klasse `Database` den Zugriff auf das DBMS durchführen können
- Mit dieser Klasse über `async` und `await` auch mehrere Datenbankoperationen hintereinander in einer Transaktion durchführen können

Installation des MySQL-Treibers für Node.js

```
$ npm install mysql
```

Treiber ist in JavaScript geschrieben und macht es nicht notwendig, zusätzliche externe Bibliotheken zu installieren

MySQL-Server muss aber auf Rechner auf dem Node.js läuft laufen

1. Variante: Direkte Ausgabe des Ergebnisses

```
const mysql = require('mysql');
const connectionProperties = {
  host: 'localhost',
  user: 'root',
  password: 'masterkey',
  database: 'movie-db'
};
function getAll() {
  const connection = mysql.createConnection(connectionProperties);
  const sql = `
    SELECT movies.id, title, year, published,
      users.username AS owner, CONCAT(users.firstname, ' ',
      users.secondname) AS fullname
      FROM movies, users
      WHERE movies.owner = users.id
      ORDER BY title;
  `;
  connection.query(sql, (error, result) => {
    connection.end();
    if (error) { throw error; }
    console.log(result);
  });
}
```

```
[
  RowDataPacket {
    id: 3,
    title: 'Capitain America',
    year: 2001,
    published: 0,
    owner: 'sepp',
    fullname: 'Sepp Hintner'
  },
  RowDataPacket {
    id: 1,
    title: 'Iron Man',
    year: 2008,
    published: 1,
    owner: 'sepp',
    fullname: 'Sepp Hintner'
  },
  RowDataPacket {
    id: 2,
    title: 'Thor',
    year: 2011,
    published: 1,
    owner: 'resi',
    fullname: 'Resi Rettich'
  }
]
```

- Verbindung wird bei `query()` geöffnet und mit `end()` geschlossen
- Das Schließen der Verbindung mit `end()` ist WICHTIG!!!
- *Callback-Methode* bei `query()` wird *asynchron* ausgeführt und zwar wenn Ergebnis bzw. Fehler vom DBMS vorliegt

PROBLEM: Gemäß MVC-Pattern muss die Ausgabe vom Controller gesteuert im View erfolgen

2. Variante: Berücksichtigung des MVC-Patterns

/movie/movie.model.js

```
const mysql = require('mysql');
const connectionProperties = { … };
function getAll(processResultCallback) {
  const connection = mysql.createConnection(connectionProperties);
  const sql = ` … `;
  connection.query(sql, (error, result) => {
    connection.end();
    processResultCallback(error, result);
  });
}
```

/movie/movie.controller.js

```
…
function listAction(request, response) {
  movieModel.getAll(
    (error, result) =>
      response.send(
        error ? movieView.renderError(error) :
        movieView.renderList(result))
  );
}
```

Callback-Funktion wird der Methode `getAll()` übergeben, welche dann diese beim Vorhandensein des Ergebnisses ausführt

PROBLEM: Unübersichtliches Zusammenspiel zw. Controller, Model und View

3. bessere Variante: Promise liefert Ergebnis an Controller zurück

/movie/movie.model.js

```
const mysql = require('mysql');
const connectionProperties = { … };
function getAllPromise() {
  return new Promise((resolve, reject) => {
    const connection = mysql.createConnection(…);
    const sql = ` … `;
    connection.query(sql, (error, result) => {
      connection.end();
      if (error) { reject(error); }
      resolve(result);
    });
  });
}
```

/movie/movie.controller.js

```
…
function listActionPromise(request, response) {
  movieModel.getAllPromise()
    .then(result => response.send(movieView.renderList(result)))
    .catch(error => response.send(movieView.renderError(error)));
  );
```

FRAGE: Angenommen es sollen innerhalb von `getAllPromises()` mehrere Queries hintereinander ausgeführt werden. Wie würden Sie dies realisieren?

PROBLEM: Das hintereinander Ausführen mehrerer Queries macht Methode unübersichtlich

4. bessere Variante: Internes Database-Objekt organisiert Zugriff

/movie/movie.model.js

```
const mysql = require('mysql');
const connectionProperties = { … };
class Database {
  constructor(connectionProperties) {
    this.connection = mysql.createConnection(…);
  }
  query(sql, params) {
    return new Promise((resolve, reject) => {
      this.connection.query(sql, params, (error, result) => {
        if (error) { reject(error); }
        resolve(result);
      });
    });
  }
  queryClose(sql, params) {
    const ret = this.query(sql, params);
    this.close();
    return ret;
  }
  close() {
    return new Promise((resolve, reject) =>
      this.connection.end(() => resolve()[1])
    );
  }
}
function getAllDatabase() {
  const database = new Database(…);
  const sql = ` … `;
  return database.queryClose(sql);
}
async function getAllDatabaseAsync() {
  try {
    const database = new Database(…);
    const sql = ` … `;
    const result = await database.queryClose(sql);
    return Promise.resolve(result);
  } catch (error) {
    return Promise.reject(error);
  }
}
```

- Am Controller müssen keine Änderungen durchgeführt werden

[1] Es wird versucht, die Datenbankverbindung zu schließen. In jedem Fall wird das Promise erfolgreich beendet, sonst würde im Fehlerfall Node.js abstürzen

- Klasse `Database` wird nicht veröffentlicht
- `params`-Array. Platzhalter im SQL-String mit ? gekennzeichnet

Beispiel: Löschen eines Benutzers mit all seinen Filmen

```
try {
  const sql1 = `START TRANSACTION;`;
  const sql2 = `DELETE movies
    FROM movies, users
    WHERE movies.owner = users.id
      AND users.username = ?;`;
  const sql3 = `DELETE
    FROM users
    WHERE username = ?;`;
  const sql4 = `COMMIT;`;
} catch (error) {
  const sql5 = `ROLLBACK;`;
}
```

```
async function deleteUser(username) {
  const database = new Database(…);
  try {
    const sql1 = ` …`;
    await database.query(sql1);
    const sql2 = ` … `;
    const result1 = await database.query(sql2, [ username ]);
    const sql3 = ` … `;
    const result2 = await database.query(sql3, [ username ]);
    const sql4 = ` … `;
    await database.queryClose(sql4);
    return Promise.resolve(
      `${result1.affectedRows} movies deleted,
       ${result2.affectedRows} users deleted`);
  } catch (error) {
    try {
      const sql5 = ` … `;
      await database.queryClose(sql5);
    } catch (error) { }
    return Promise.reject(error);
  }
}
```

`affectedRows`	Anzahl der eingefügten/geänderten/gelöschten Datensätze
`insertId`	Primärschlüsselwert des neu eingefügten Datensatzes