LU17 – CRUD mit Express & MySQL (Tabelle ''post'')

Learning Objectives

- Sie können eine **MySQL-Datenbank** aus Ihrem Express-Server heraus ansprechen.
- Sie können CRUD-Routen für `posts` so erweitern, dass sie mit einer **echten Tabelle `post`** arbeiten.
- Sie setzen einfache **Validierung** ein (Pflichtfelder prüfen).
- Sie verwenden passende **HTTP-Statuscodes** (200, 201, 400, 404, 500).
- Sie testen alle Operationen mit **Postman**.

Ausgangslage

In LU16 haben Sie eine kleine API für Social-Media-Posts gebaut:

- In-Memory-Liste `posts` im Code,
- Routen wie:
    1. `GET /posts`
    2. `GET /posts/:id`
    3. `POST /posts`

Jetzt machen wir einen wichtigen Schritt in Richtung „echte“ Anwendung:

- Die Daten kommen nicht mehr aus einer JS-Liste,
- sondern aus einer **MySQL-Tabelle `post`**,
- Ihr Express-Server wird zur **Brücke** zwischen Client (Postman/Browser) und Datenbank.

Vorbereitung: MySQL-Tabelle ''post''

Eine mögliche Tabellenstruktur (vereinfacht):

Spalte	**Datentyp**	**Beschreibung**
`post_id`	INT, PK, AI	Primärschlüssel
`user_id`	INT	Referenz auf User (FK)
`title`	VARCHAR(255)	Titel des Posts
`image_url`	VARCHAR(512)	Bild-URL
`description`	TEXT	Beschreibungstext
`likes`	INT	Anzahl Likes

Diese Tabelle können Sie wie gewohnt mit einem SQL-DDL-Skript anlegen. (Entweder vorgegeben im Unterricht oder von Ihnen selbst erstellt.)

MySQL aus Node.js ansprechen

Wir verwenden das Paket **`mysql2`** für die Verbindung:

1. Installation von mysql2

Im Projektordner (Terminal in WebStorm):

```
npm install mysql2
```

Dadurch wird die Bibliothek in `node_modules` installiert und in `package.json` als Abhängigkeit eingetragen.

2. Verbindung zur Datenbank herstellen

Wir erweitern unseren Server aus LU16 um eine Datenbankverbindung. Alles bleibt weiterhin in einer Datei (z.B. `index.js`), um es für das Modul einfach zu halten.

```
import express from 'express';
import mysql from 'mysql2';

const app = express();
const port = 3000;

app.use(express.json());

// Verbindung zur MySQL-Datenbank
const db = mysql.createConnection({
  host: 'localhost',
  database: 'social_media',        // Ihre DB
  user: 'restrictedUser',          // AppUser, nicht root
  password: 'SafePassword123'
});

db.connect((err) => {
  if (err) {
    console.error('DB-Verbindung fehlgeschlagen:', err);
    return;
  }
  console.log('Mit MySQL verbunden');
});
```

```
// Test-Route
app.get('/', (req, res) => {
  res.send('API mit Datenbank ist online');
});

app.listen(port, () => {
  console.log(`API läuft auf http://localhost:${port}`);
});
```

Wenn Sie den Server starten (`npm run dev`) und im Terminal **„Mit MySQL verbunden“** sehen, ist die Verbindung erfolgreich.

READ – Posts aus der Datenbank holen

Jetzt ersetzen wir die In-Memory-Liste durch echte SELECT-Abfragen.

1. Alle Posts – "GET /posts"

```
// READ – alle Posts aus der DB
app.get('/posts', (req, res) => {
  const sql = `
    SELECT post_id, user_id, title, image_url, description,
likes
    FROM post
  `;

  db.query(sql, (err, results) => {
    if (err) {
      console.error('DB-Fehler bei GET /posts:', err);
      return res.status(500).send('Serverfehler bei der
Post-Abfrage');
    }

    res.status(200).json(results);
  });
});
```

Test mit Postman

- Methode: GET
- URL: http://localhost:3000/posts
- Erwartung:
    1. Status 200 OK

2. JSON-Array mit allen Zeilen aus der Tabelle `post`

2. Einzelner Post – "GET /posts/:id"

```
// READ – einzelner Post nach ID aus der DB
app.get('/posts/:id', (req, res) => {
  const id = Number(req.params.id);

  const sql = `
    SELECT post_id, user_id, title, image_url, description,
likes
    FROM post
    WHERE post_id = ?
  `;

  db.query(sql, [id], (err, results) => {
    if (err) {
      console.error('DB-Fehler bei GET /posts/:id:', err);
      return res.status(500).send('Serverfehler bei der
Post-Abfrage');
    }

    if (results.length === 0) {
      return res.status(404).send('Post nicht gefunden');
    }

    res.status(200).json(results[0]);
  });
});
```

CREATE – Neuen Post in der DB speichern

Jetzt bauen wir die POST `/posts`-Route so um, dass sie **`INSERT`** in die MySQL-Tabelle ausführt.

```
// CREATE – neuen Post in der DB anlegen
app.post('/posts', (req, res) => {
  const { user_id, title, image_url, description } =
req.body;

  // Einfache Validierung der Pflichtfelder
  if (!user_id || !title) {
    return res.status(400).send('Bitte mindestens user_id
und title angeben.');
```

```
  }

  const sql = `
    INSERT INTO post (user_id, title, image_url,
description, likes)
    VALUES (?, ?, ?, ?, 0)
  `;

  const values = [
    user_id,
    title,
    image_url || '',
    description || ''
  ];

  db.query(sql, values, (err, result) => {
    if (err) {
      console.error('DB-Fehler bei POST /posts:', err);
      return res.status(500).send('Serverfehler beim
Erstellen des Posts');
    }

    // Neu erzeugte ID steht in result.insertId
    const newPost = {
      post_id: result.insertId,
      user_id,
      title,
      image_url: image_url || '',
      description: description || '',
      likes: 0
    };

    res.status(201).json(newPost);
  });
});
```

Test mit Postman

- Methode: POST
- URL: http://localhost:3000/posts
- Body (JSON):

```
{
  "user_id": 1,
  "title": "Neuer DB-Post",
  "image_url": "https://example.com/post.jpg",
  "description": "Dieser Post wurde in MySQL gespeichert."
```

```
}
```

- Erwartung:
  1. Status `201 Created`
  2. JSON-Objekt mit neuer `post_id`
  3. Eintrag ist in der Tabelle `post` sichtbar

UPDATE – Bestehenden Post ändern (PUT /posts/:id)

Wir implementieren jetzt ein UPDATE mit einfacher Validierung.

```
// UPDATE – vorhandenen Post aktualisieren
app.put('/posts/:id', (req, res) => {
  const id = Number(req.params.id);
  const { title, image_url, description, likes } = req.body;

  // Mindestens ein Feld muss geändert werden
  if (!title && !image_url && !description && likes ===
undefined) {
    return res.status(400).send(
      'Bitte mindestens eines der Felder title, image_url,
description oder likes angeben.'
    );
  }

  const sql = `
    UPDATE post
    SET
      title = COALESCE(?, title),
      image_url = COALESCE(?, image_url),
      description = COALESCE(?, description),
      likes = COALESCE(?, likes)
    WHERE post_id = ?
  `;

  const values = [
    title ?? null,
    image_url ?? null,
    description ?? null,
    likes ?? null,
    id
  ];

  db.query(sql, values, (err, result) => {
    if (err) {
```

```
      console.error('DB-Fehler bei PUT /posts/:id:', err);
      return res.status(500).send('Serverfehler beim
Aktualisieren des Posts');
    }

    if (result.affectedRows === 0) {
      return res.status(404).send('Post nicht gefunden');
    }

    res.status(200).send(`Post mit ID ${id} wurde
aktualisiert.`);
  });
});
```

Hinweise zur Validierung

- Wenn gar keine Felder im Body sind → `400 Bad Request`
- Wenn die ID in der DB nicht existiert → `404 Not Found`
- Bei Datenbankfehler → `500 Internal Server Error`

DELETE – Post löschen (DELETE /posts/:id)

Zum Schluss implementieren wir DELETE:

```
// DELETE – Post löschen
app.delete('/posts/:id', (req, res) => {
  const id = Number(req.params.id);

  const sql = 'DELETE FROM post WHERE post_id = ?';

  db.query(sql, [id], (err, result) => {
    if (err) {
      console.error('DB-Fehler bei DELETE /posts/:id:',
err);
      return res.status(500).send('Serverfehler beim Löschen
des Posts');
    }

    if (result.affectedRows === 0) {
      return res.status(404).send('Post nicht gefunden');
    }

    res.status(200).send(`Post mit ID ${id} wurde
gelöscht.`);
  });
});
```

Zusammenfassung & Transfer auf Ihr Projekt

In dieser Learning Unit haben Sie:

- Ihren Express-Server mit **MySQL** verbunden (`mysql2`),
- **CRUD-Routen** für `posts` so erweitert, dass sie auf die echte Tabelle `post` zugreifen,
- einfache **Validierung** eingebaut (Pflichtfelder, leere Updates),
- sinnvolle **HTTP-Statuscodes** verwendet:
    1. `201` für erfolgreiches Erstellen,
    2. `200` für erfolgreiche Lese-/Update-/Delete-Operationen,
    3. `400` bei falschen Eingaben,
    4. `404` bei nicht gefundenen Datensätzen,
    5. `500` bei Server-/Datenbankfehlern.

Nächster Schritt für Ihr Modulprojekt:

- Erstellen Sie für Ihren eigenen Use Case (Reisen, Filme, Bücher, …) eine entsprechende Tabelle (oder mehrere Tabellen).
- Bauen Sie ähnliche Routen:
    1. `GET /[ressource]`, `GET /[ressource]/:id`
    2. `POST /[ressource]`
    3. `PUT /[ressource]/:id`
    4. `DELETE /[ressource]/:id`
- Testen Sie alles mit **Postman**.
- Nutzen Sie diese Struktur als Grundlage für die LB3-Projektarbeit.

Vocabulary

English	Deutsch
query	Abfrage (SQL)
result set	Resultatmenge (Abfrage-Ergebnis)
to insert	einfügen
to update	aktualisieren
to delete	löschen
error handling	Fehlerbehandlung
validation	Validierung / Eingabeprüfung