

LU17 - CRUD mit Express & MySQL

Lernziele

- Sie können Ihre **MySQL-Datenbank** aus einem Express-Server heraus ansprechen.
- Sie können die CRUD-Routen aus LU16 (GET, POST, PUT, DELETE) 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**.

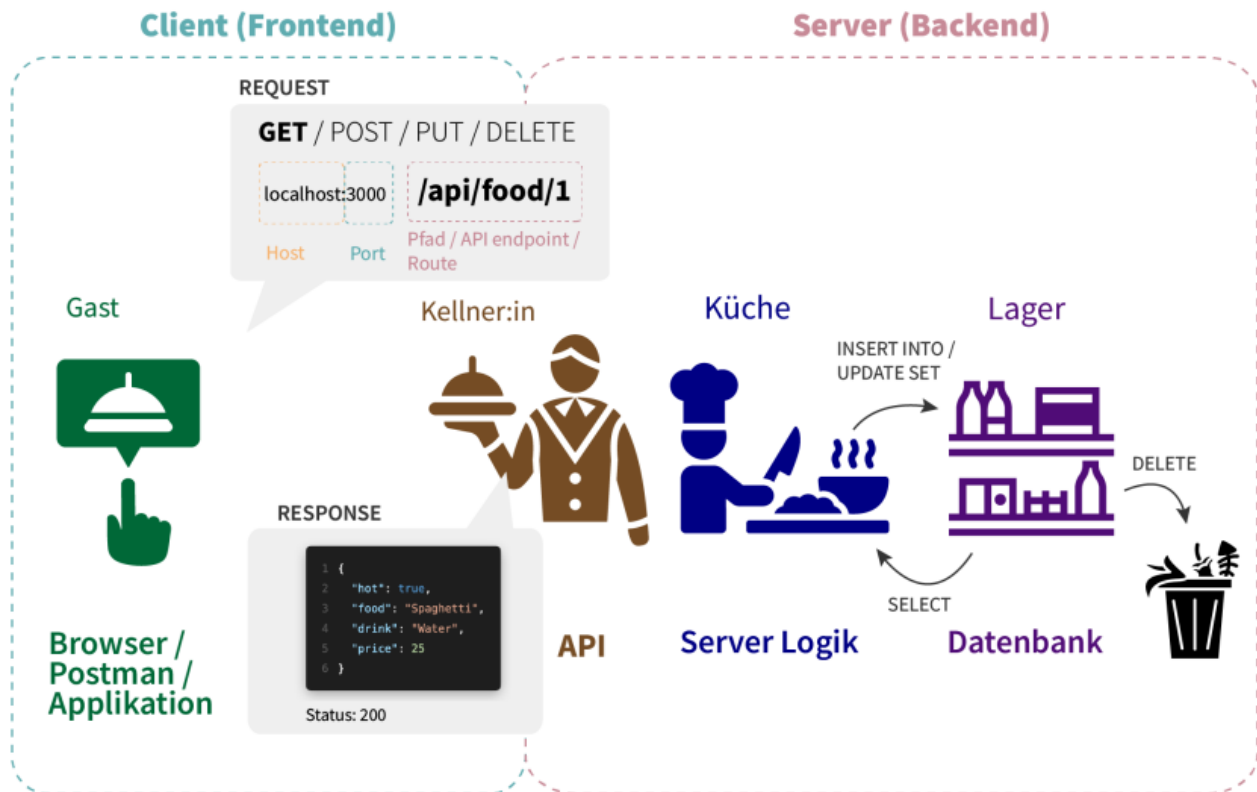
Rückblick auf LU16

In LU16 haben Sie eine API für Social-Media-Posts gebaut – aber noch ohne Datenbank:

- Daten waren in einer **JavaScript-Liste posts** gespeichert (In-Memory).
- Ihre Routen waren z.B.:
 - GET /api/posts
 - GET /api/posts/:id
 - POST /api/posts

In LU17 ersetzen Sie diese In-Memory-Liste durch eine echte Datenbank:

- Die Daten kommen aus der **MySQL-Tabelle post**.
- Ihr Express-Server ist die **Brücke** zwischen Client (Postman) und Datenbank.
- Die gleiche Idee bleibt: **HTTP-Anfrage → Route → Logik/SQL → HTTP-Antwort**.








HTTP-Statuscodes

Wenn Sie mit Postman testen, sehen Sie immer einen Statuscode. Dieser Code ist Ihr „Kurzfazit“, ob der Request geklappt hat.

HTTP-STATUSCODES

Dreistellige Codes in HTTP-Antworten, die zeigen, was mit der Anfrage passiert ist.
Die Statuscodes werden vom Server zurück an den Client gesendet.

| | | | | |
|---|--|--|---|--|
|  |  |  |  |  |
| 100-199 INFORMATIONAL (INFORMATION) | 200-299 SUCCESS (ERFOLG) | 300-399 REDIRECTION (WEITERLEITUNG) | 400-499 CLIENT ERRORS (CLIENT-FEHLER) | 500-599 SERVER ERRORS (SERVER-FEHLER) |
| <p>BEDEUTUNG: Anfrage erhalten — es geht weiter / Protokollwechsel.</p> <ul style="list-style-type: none"> • 100 Continue sende den Rest der Anfrage • 101 Switching Protocols Wechsel zu einem anderen Protokoll • 103 Early Hints Ladehinweise vor der finalen Antwort | <p>BEDEUTUNG: Anfrage erfolgreich.</p> <ul style="list-style-type: none"> • 200 OK Anfrage erfolgreich • 201 Created neue Ressource erstellt • 202 Accepted angenommen; Verarbeitung später • 204 No Content Erfolg; kein Response-Body • 205 Reset Content Erfolg; Seite/Formular zurücksetzen | <p>BEDEUTUNG: Du musst woanders hin (neue URL / andere Abrufart).</p> <ul style="list-style-type: none"> • 300 Multiple Choices mehrere Optionen verfügbar • 301 Moved Permanently permanente Weiterleitung (Link aktualisieren) • 302 Found temporäre Weiterleitung • 303 See Other Weiterleitung; Abruf per GET • 307 Temporary Redirect temporäre Weiterleitung; Methode/Body beibehalten • 308 Permanent Redirect permanent Weiterleitung; Methode/Body beibehalten | <p>BEDEUTUNG: Problem mit der Anfrage (clientseitig).</p> <ul style="list-style-type: none"> • 400 Bad Request fehlerhafte Anfrage / ungültige Daten • 401 Unauthorized nicht authentifiziert • 402 Payment Required reserviert / selten genutzt • 403 Forbidden Zugriff verboten • 404 Not Found Ressource nicht gefunden • 405 Method Not Allowed Methode hier nicht erlaubt • 408 Request Timeout Anfrage dauerte zu lange • 410 Gone dauerhaft entfernt • 429 Too Many Requests Rate-Limit (zu viele Anfragen) | <p>BEDEUTUNG: Server kann eine gültige Anfrage nicht verarbeiten.</p> <ul style="list-style-type: none"> • 500 Internal Server Error interner Serverfehler • 501 Not Implemented nicht implementiert • 502 Bad Gateway ungültige Antwort vom Upstream • 503 Service Unavailable überlastet / Wartung • 504 Gateway Timeout Upstream-Timeout • 505 HTTP Version Not Supported HTTP-Version nicht unterstützt |

| Situation | Typischer Statuscode | Bedeutung |
|--|----------------------|---|
| Erfolgreich Daten gelesen (z.B. GET) | 200 | OK (Daten kommen zurück) |
| Erfolgreich Datensatz erstellt (POST) | 201 | Created (Neuer Datensatz wurde erstellt) |
| Erfolgreich aktualisiert (PUT) | 200 | OK (Update hat geklappt) |
| Erfolgreich gelöscht (DELETE) | 200 | OK (Löschen hat geklappt; Sie können zur Kontrolle z.B. das gelöschte Objekt oder eine Message zurückgeben) |
| Pflichtfelder fehlen / ungültige Eingabe | 400 | Bad Request (Client hat falsche/fehlende Daten geschickt) |
| ID existiert nicht | 404 | Not Found (Datensatz wurde nicht gefunden) |
| Datenbank-/Serverfehler | 500 | Internal Server Error (Problem auf Server/DB-Seite) |

Vorbereitung: MySQL-Tabelle post

Eine mögliche Tabellenstruktur (vereinfacht):

| Spalte | Datentyp | Beschreibung |
|-------------|----------------------------------|------------------------|
| post_id | INT, PRIMARY KEY, AUTO_INCREMENT | Primärschlüssel |
| user_id | INT | Referenz auf User (FK) |
| title | VARCHAR(255) | Titel des Posts |
| image_url | VARCHAR(512) | Bild-URL |
| description | TEXT | Beschreibung |
| likes | INT | Anzahl Likes |

Diese Tabelle legen Sie mit einem SQL-DDL-Skript an (wie bisher im Modul). Importieren Sie zusätzlich Startdaten (DML), damit Sie direkt testen können.

Schritt 1: mysql2 installieren

Damit Node.js mit MySQL sprechen kann, verwenden wir mysql2.

```
npm install mysql2
```

Schritt 2: connect.js erstellen (DB-Verbindung auslagern)

Damit `index.js` übersichtlich bleibt, erstellen Sie eine eigene Datei `connect.js`. Diese Datei erstellt **eine Verbindung** und exportiert sie, damit Sie sie in Ihren Routes mit `db.query(...)` verwenden können.

```
import mysql from 'mysql2';

// Hinweis: Verwenden Sie hier Ihren AppUser (nicht root).
// Port 3306 ist der Standard-Port für MySQL auf localhost.
const db = mysql.createConnection({
  host: 'localhost',
  user: 'app_user',
  password: 'YOUR_PASSWORD_HERE',
  port: 3306,
  database: 'social_media',
});
```

```
// connect() öffnet die Verbindung und gibt bei
// Erfolg/Fehler eine Meldung aus.
db.connect((err) => {
  if (err) {
    console.error('DB-Verbindung fehlgeschlagen:', err);
    return;
  }
  console.log('Mit MySQL verbunden');
});

export { db };
```

Was passiert hier?

- db ist die offene Verbindung und kann in index.js importiert werden (z.B. import { db } from „./connect.js“;).
- In der LB03 sollen Sie **nicht** als root arbeiten → verwenden Sie einen AppUser.

Schritt 3: index.js vorbereiten (Express + DB verwenden)

Sie bauen auf dem Setup aus LU16 auf:

- app.use(express.json()) bleibt wichtig (für req.body).
- Sie importieren db aus connect.js. Die Verbindung wird in connect.js bereits aufgebaut (db.connect(...)).
- Löschen Sie die Test-Tabelle aus LU16 let posts = [{ post_id: 1, ... }];

```
import express from 'express';
import { db } from './connect.js';

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

app.use(express.json());

// Test-Route: schnell prüfen, ob der Server läuft
app.get('/', (req, res) => {
  res.send('API mit Datenbank ist online');
});

// Optional: kleine Test-Route für DB (zeigt Serverzeit aus
// MySQL)
app.get('/api/db-test', (req, res) => {
  db.query('SELECT NOW() AS server_time', (err, results) => {
    if (err) {
```

```
    console.error('DB-Fehler bei /api/db-test:', err);
    return res.status(500).send('DB-Test fehlgeschlagen');
  }
  res.status(200).json(results[0]);
});
});

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

Was passiert hier?

- db kommt aus connect.js und ist die MySQL-Verbindung.
- Ihre Routen können direkt db.query(...) verwenden.
- Mit /api/db-test prüfen Sie schnell, ob die DB erreichbar ist (Status 200 + JSON), ohne schon CRUD zu programmieren.

Server starten Nicht alle Lernenden haben nodemon installiert. Beides ist möglich:

```
node index.js
```

oder (falls Sie ein dev-Script mit nodemon haben):

```
npm run dev
```

CRUD-Routen: posts jetzt mit echter MySQL-Tabelle

Ab hier ersetzen Sie die In-Memory-Liste aus LU16 durch SQL.

Konvention: Wir bleiben wie in LU16 bei /api/posts (statt nur /posts), damit Ihre API klar erkennbar ist.

READ: alle Posts (GET /api/posts)

```
// READ – alle Posts aus der DB
// Route: GET http://localhost:3000/api/posts
app.get('/api/posts', (req, res) => {

  // SQL-Abfrage: alle Spalten, die wir zurückgeben möchten
  const sql = `
    SELECT post_id, user_id, title, image_url, description,
    likes
    FROM post
  `;

  // db.query(...) führt die SQL-Abfrage aus
  db.query(sql, (err, results) => {

    // Falls die DB einen Fehler liefert -> 500
    if (err) {
      console.error('DB-Fehler bei GET /api/posts:', err);
      return res.status(500).send('Serverfehler bei der
Post-Abfrage');
    }

    // results ist ein Array mit Zeilen (Rows)
    res.status(200).json(results);
  });
});
```

Test in Postman

- Methode: GET
- URL: <http://localhost:3000/api/posts>
- Erwartung: 200 + JSON-Liste

READ: einzelner Post (GET /api/posts/:id)

```
// READ – einzelner Post nach ID aus der DB
// Route: GET http://localhost:3000/api/posts/1
app.get('/api/posts/:id', (req, res) => {

  // Route-Parameter :id kommt immer als String ->
  Number(...) macht eine Zahl daraus
  const id = Number(req.params.id);

  // Einfache Validierung: ist id überhaupt eine Zahl?
  if (Number.isNaN(id)) {
    return res.status(400).send('Ungültige ID (muss eine
Zahl sein)');
  }
});
```

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

// Platzhalter ? wird durch id ersetzt -> Schutz vor SQL-
Injection
db.query(sql, [id], (err, results) => {

  if (err) {
    console.error('DB-Fehler bei GET /api/posts/:id:',
err);
    return res.status(500).send('Serverfehler bei der
Post-Abfrage');
  }

  // Wenn keine Zeile gefunden -> 404
  if (results.length === 0) {
    return res.status(404).send('Post nicht gefunden');
  }

  // results[0] ist der erste (und hier einzige) Treffer
  res.status(200).json(results[0]);
});
});
```

CREATE: neuen Post erstellen (POST /api/posts)

```
// CREATE – neuen Post in der DB anlegen
// Route: POST http://localhost:3000/api/posts
app.post('/api/posts', (req, res) => {

  // Daten kommen aus dem Request-Body (Postman: Body -> raw
-> JSON)
  const user_id = req.body.user_id;
  const title = req.body.title;
  const image_url = req.body.image_url;
  const description = req.body.description;

  // Validierung: Pflichtfelder
  if (user_id === undefined || user_id === null || title ===
undefined || title === null || 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 || ''
];

// INSERT ausführen
db.query(sql, values, (err, result) => {

  if (err) {
    console.error('DB-Fehler bei POST /api/posts:', err);
    return res.status(500).send('Serverfehler beim
Erstellen des Posts');
  }

  // insertId kommt von MySQL AUTO_INCREMENT
  const newPost = {
    post_id: result.insertId,
    user_id,
    title,
    image_url: image_url || '',
    description: description || '',
    likes: 0
  };

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

Test-Body (Postman → Body → raw → JSON)

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

UPDATE: Post ändern (PUT /api/posts/:id)

Hier übernehmen Sie die Idee aus LU16 (Update) – aber jetzt mit SQL UPDATE.

```
// UPDATE – Post vollständig ersetzen (PUT)
app.put('/api/posts/:id', (req, res) => {
  const id = Number(req.params.id);
  if (Number.isNaN(id)) {
    return res.status(400).send('Ungültige ID (muss eine Zahl sein)');
  }

  // Alle Felder werden erwartet
  const user_id = req.body.user_id;
  const title = req.body.title;
  const image_url = req.body.image_url;
  const description = req.body.description;
  const likes = req.body.likes;

  // Validierung: alle Pflichtfelder vorhanden?
  if (user_id === undefined || title === undefined ||
    image_url === undefined || description === undefined ||
    likes === undefined) {
    return res.status(400).send('Bitte user_id, title, image_url, description und likes mitsenden (PUT ersetzt alles).');
  }

  const sql = UPDATE post SET user_id = ?, title = ?,
    image_url = ?, description = ?, likes = ? WHERE post_id = ?
  ;

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

  db.query(sql, values, (err, result) => {
    if (err) {
      console.error('DB-Fehler bei PUT /api/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');
    }
  })
}
```

```
// Update hat geklappt: Status 200 + das "neue" Objekt zurückgeben
res.status(200).json({
  post_id: id,
  user_id,
  title,
  image_url,
  description,
  likes
});

});
});
```

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

```
// DELETE – Post löschen
// Route: DELETE http://localhost:3000/api/posts/1
app.delete('/api/posts/:id', (req, res) => {

  const id = Number(req.params.id);

  if (Number.isNaN(id)) {
    return res.status(400).send('Ungültige ID (muss eine Zahl sein)');
  }

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

  db.query(sql, [id], (err, result) => {

    if (err) {
      console.error('DB-Fehler bei DELETE /api/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');
    }

    // Statuscode: 200 + Message
    res.status(200).json({ message: `Post mit post_id=${id} wurde gelöscht.` });

  });
});
```

```
});
```

Typische Fehlerquellen (aus LU16, jetzt noch wichtiger)

- `req.params.id` ist immer ein String → mit `Number(...)` umwandeln (und `Number.isNaN(...)` prüfen).
- Ohne `app.use(express.json())` ist `req.body` leer.
- SQL immer mit Platzhaltern ? schreiben (Prepared Statements).
- Denken Sie bei `db.query(...)` immer an die drei Fälle:
 - Eingaben ok? → sonst 400
 - Datensatz gefunden? → sonst 404 (`results.length === 0` oder `affectedRows === 0`)
 - DB-Fehler? → 500

Transfer auf Ihr Projekt (LB03)

Für Ihr Projekt ersetzen Sie `post` durch Ihre eigenen Tabellen (z.B. `serie`, `actor`, `serie_actor`):

- Ressourcen-Route: z.B. `/api/serien`
- CRUD: GET, POST, PUT, DELETE
- Zusätzlich:
 - mindestens **eine JOIN-Route** (z.B. Serien inkl. Schauspieler:innen)
 - mindestens **eine Aggregat-Route** (z.B. Durchschnittsbewertung pro Genre)

From:
<https://wiki.bzz.ch/> - BZZ - Modulwiki

Permanent link:
https://wiki.bzz.ch/modul/m290_guko/learningunits/lu17/theorie/a_intro?rev=1766759478

Last update: **2025/12/26 15:31**

