LU16 – HTTP, CRUD & Postman (ohne Datenbank)

Learning Objectives

- Sie können grob erklären, wie eine **HTTP-Anfrage** und eine **HTTP-Antwort** funktionieren.
- Sie können die vier wichtigsten HTTP-Methoden **GET, POST, PUT, DELETE** den CRUD-Operationen zuordnen.
- Sie verstehen, was eine **API** und was eine **Route** in Express ist.
- Sie können mit **Postman** einfache Requests an Ihren Express-Server schicken.
- Sie können erste **CRUD-Routen** für `posts` mit einer einfachen In-Memory-Liste erstellen (ohne Datenbank).

Von SQL-CRUD zu Web-CRUD

Bisher im Modul M290:

- Sie haben in **MySQL** Tabellen erstellt, Daten importiert und Abfragen geschrieben.
- Sie kennen **CRUD**:
  1. **C**reate → `INSERT`
  2. **R**ead → `SELECT`
  3. **U**pdate → `UPDATE`
  4. **D**elete → `DELETE`

Jetzt übertragen wir diese Idee auf das **Web**: Statt direkt SQL zu tippen, schicken wir **HTTP-Anfragen** an einen **Backend-Server (Express)**, und dieser Server führt für uns die passenden CRUD-Operationen aus.

Wir bauen zuerst eine **Mini-API für Social-Media-Posts**, noch ohne Datenbank – nur mit Daten im Speicher. So können Sie in Ruhe **HTTP, Routen und Postman** verstehen, bevor wir in LU17 MySQL anschliessen.

HTTP in einfachen Worten – Restaurant-Vergleich

Stellen Sie sich das so vor:

- **Client** (Browser oder Postman) = Gast im Restaurant
- **Server** (Express/Node.js) = Küche
- **HTTP-Anfrage** = Bestellung, die der Kellner zur Küche bringt
- **HTTP-Antwort** = fertiges Gericht, das der Kellner zurückbringt

Jede HTTP-Anfrage hat unter anderem:

- eine **Methode** (z.B. GET, POST, PUT, DELETE),
- einen **Pfad** (z.B. `/posts`, `/posts/3`),

- optional einen **Body** (z.B. JSON-Daten bei POST/PUT).

Der Server reagiert auf eine Anfrage, indem er:

```
1. eine passende **Route** findet (z.B. ''app.get('/posts', ...)''),
2. etwas ausführt (z.B. Daten holen oder speichern),
3. genau **eine** Antwort zurückschickt (z.B. JSON oder Text).
```

CRUD & HTTP-Methoden

CRUD-Operationen werden bei Web-APIs typischerweise folgenden HTTP-Methoden zugeordnet:

CRUD	HTTP-Methode	Typisches Beispiel einer Route
Create	**POST**	POST `/posts` → Neuer Post wird angelegt
Read	**GET**	GET `/posts` → Alle Posts anzeigen; GET `/posts/5` → Post mit ID 5 anzeigen
Update	**PUT**	PUT `/posts/5` → Post mit ID 5 vollständig aktualisieren
Delete	**DELETE**	DELETE `/posts/5` → Post mit ID 5 löschen

Später werden Sie für Ihre eigenen Use Cases (Reisen, Filme, Bücher, …) genau solche Routen definieren, z.B. GET `/trips`, POST `/books`, usw.

Postman – unser „Frontend-Ersatz“

Im Modul M290 programmieren wir **kein eigenes Browser-Frontend**. Stattdessen benutzen wir **Postman** als Client:

- Postman kann GET, POST, PUT, DELETE-Requests an Ihre API senden.
- Sie sehen direkt:
  1. den **Statuscode** (z.B. 200, 201, 400, 404, 500),
  2. die **Response-Header**,
  3. den **Response-Body** (Text oder JSON).
- Sie können im Body bequem JSON eingeben (z.B. neuen `post` anlegen).

So können Sie Ihre API testen, als wäre schon ein fertiges Frontend vorhanden – nur viel einfacher und kontrollierter.

Beispiel-API: Social-Media-Posts (ohne Datenbank)

Wir verwenden das vereinfachte `post`-Modell aus Ihrem Social-Media-ERD:

- `post_id` (PK, Nummer)
- `user_id` (FK zum User – hier nur als Zahl)
- `title` (Titel des Posts)

- `image_url` (Bild-URL)
- `description` (Beschreibungstext)
- `likes` (Anzahl Likes)

Ziel von LU16:

- Einen kleinen Express-Server mit einer **In-Memory-Liste** von Posts erstellen.
- Erste Routen:
  1. `GET /posts` → alle Posts
  2. `GET /posts/:id` → ein Post
  3. `POST /posts` → neuen Post hinzufügen

1. Start: Minimale Express-API für Posts

Wir bauen auf Ihrem bekannten Grundserver aus LU15 auf.

```
import express from 'express';

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

// Middleware: JSON-Body einlesen (für POST/PUT)
app.use(express.json());

// "Fake-Datenbank" im Arbeitsspeicher
let posts = [
  {
    post_id: 1,
    user_id: 1,
    title: 'Erster Post',
    image_url: 'https://example.com/image1.jpg',
    description: 'Hallo Welt aus unserem Backend!',
    likes: 10
  },
  {
    post_id: 2,
    user_id: 2,
    title: 'Zweiter Post',
    image_url: 'https://example.com/image2.jpg',
    description: 'Noch ein Test-Post.',
    likes: 5
  }
];

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

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

Wenn Sie diesen Server mit `npm run dev` starten, sollten Sie im Browser unter http://localhost:3000/ den Text **„API ist online“** sehen.

2. READ – Alle Posts abfragen: ''GET /posts''

Jetzt fügen wir eine Route hinzu, die alle Posts zurückliefert:

```
// READ – alle Posts
app.get('/posts', (req, res) => {
  res.status(200).json(posts);
});
```

Test mit Postman oder Browser

- Methode: GET
- URL: http://localhost:3000/posts
- Erwartung:
  1. Status `200 OK`
  2. JSON-Array mit den zwei Beispiel-Posts

3. READ – Einzelnen Post nach ID abfragen: ''GET /posts/:id''

Wir möchten einen einzelnen Post anhand der `post_id` holen.

```
// READ – einzelner Post nach ID
app.get('/posts/:id', (req, res) => {
  const id = Number(req.params.id);     // Pfad-Parameter
holen und in Zahl umwandeln
  const post = posts.find(p => p.post_id === id);

  if (!post) {
    return res.status(404).send('Post nicht gefunden');
  }

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

Test mit Postman

- Methode: GET
- URL: `http://localhost:3000/posts/1`
- Erwartung:
  1. Status `200 OK`
  2. JSON-Objekt mit `post_id: 1`
- URL: `http://localhost:3000/posts/999`
  1. Status `404 Not Found`
  2. Body: `Post nicht gefunden`

4. CREATE – Neuen Post anlegen: "POST /posts"

Nun soll ein neuer Post erstellt werden. Dazu schicken wir einen JSON-Body mit den notwendigen Feldern.

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

  // ganz einfache Validierung
  if (!user_id || !title) {
    return res.status(400).send('Bitte mindestens user_id
und title angeben.');
  }

  // neue ID bestimmen (hier: max + 1)
  const newId = posts.length > 0 ? Math.max(...posts.map(p
=> p.post_id)) + 1 : 1;

  const newPost = {
    post_id: newId,
    user_id,
    title,
    image_url: image_url || '',
    description: description || '',
    likes: 0
  };

  posts.push(newPost);

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

Test mit Postman

```
1. Methode: ''POST''
2. URL: ''http://localhost:3000/posts''
3. Tab ''Body'' → ''raw'' → ''JSON'' auswählen
4. Beispiel-Body:
```

```
{
  "user_id": 3,
  "title": "Mein erster echter Post",
  "image_url": "https://example.com/image3.jpg",
  "description": "Gerade mit Postman erstellt!"
}
```

```
5. ''Send'' klicken
   * Erwartung:
     - Status ''201 Created''
     - JSON-Objekt mit neuer ''post_id'' (z.B. 3)
6. Anschliessend ''GET /posts'' erneut ausführen → der neue Post sollte in
der Liste sein.
```

Ausblick auf LU17

In dieser Learning Unit haben Sie:

- die **HTTP-Methoden** im Kontext von CRUD kennengelernt,
- mit **Postman** einfache Requests an Ihren Express-Server geschickt,
- eine kleine API mit einer In-Memory-Liste von `posts` erstellt (`GET /posts`, `GET /posts/:id`, `POST /posts`).

In **LU17**:

- ersetzen wir die In-Memory-Liste durch eine **echte MySQL-Tabelle `post`**,
- ergänzen **Update (PUT)** und **Delete (DELETE)** für vollständiges CRUD,
- bauen einfache **Validierung** und **Fehlerbehandlung** (HTTP-Statuscodes) ein,
- damit Sie diese Struktur für Ihren eigenen Projekt-Use-Case übernehmen können.

Vocabulary

English	**Deutsch**
request	Anfrage
response	Antwort
endpoint / route	Endpunkt / Route
status code	Statuscode
body (request body)	(Request-)Body, Datenkörper
to validate	validieren, überprüfen

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m290_guko/learningunits/lu16/theorie/a_intro?rev=1765380370

Last update: **2025/12/10 16:26**