LU16 – HTTP, CRUD & Postman

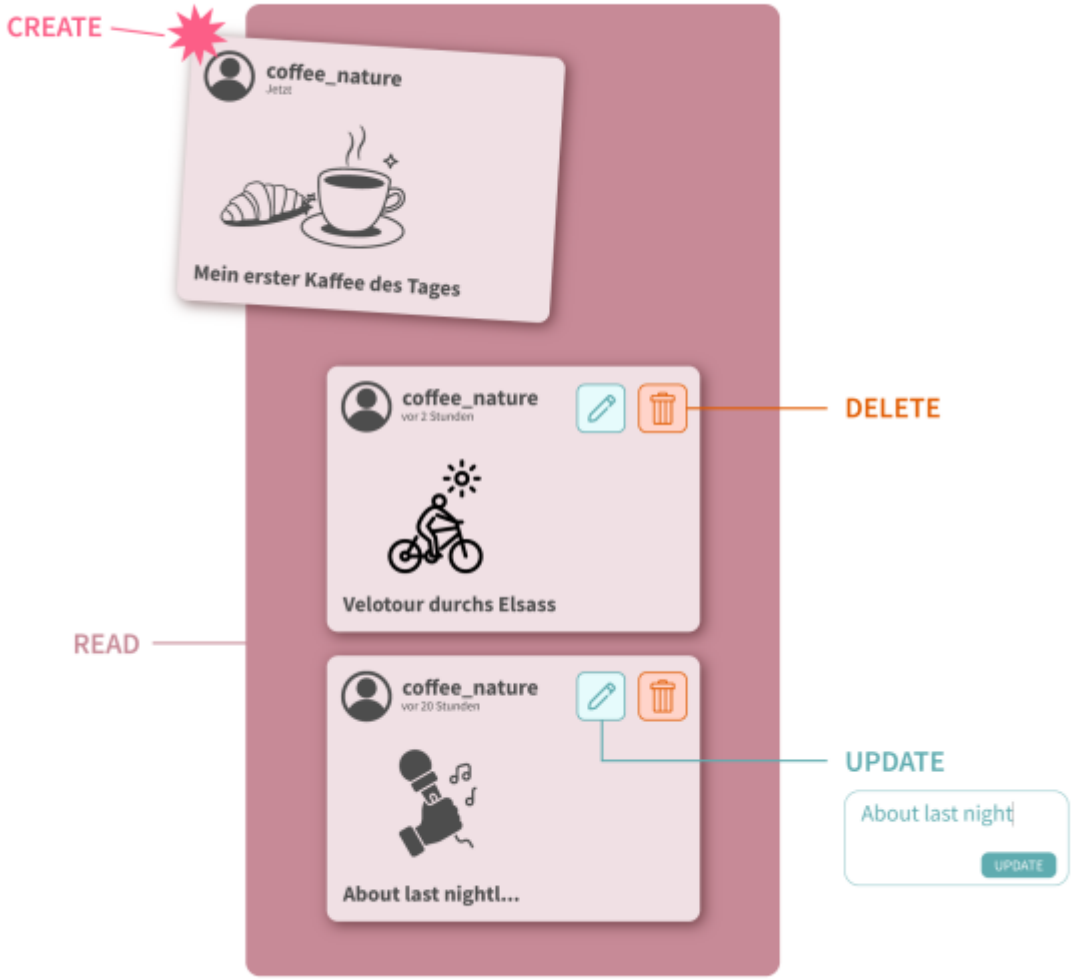
Lernziele

- Sie können erklären, wie eine **HTTP-Anfrage** und eine **HTTP-Antwort** funktionieren. [1)]
- 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 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**:
  - **C**reate → `INSERT`
  - **R**ead → `SELECT`
  - **U**pdate → `UPDATE`
  - **D**elete → `DELETE`



Jetzt übertragen wir diese Idee auf das **Web**: Statt direkt SQL zu tippen, schicken wir **HTTP-Anfragen** an einen **Backend-Server** [2)] – bei uns: **Express (Node.js)**. Der Server führt für uns die passenden CRUD-Operationen aus und sendet eine Antwort zurück.

Wir bauen zuerst eine **API für Social-Media-Posts**, noch ohne Datenbank – nur mit Daten in einem Javascript-Objekt. So können wir **HTTP, Routen und Postman** kennenlernen, bevor wir in in der nächsten Unterrichtseinheit MySQL an Express anbinden.

Schauen Sie sich das Video zum Unterricht an – Einführung in APIs und Umsetzung von CRUD mit express: Einführungs-Video API/CRUD

HTTP-Methoden – wie Bestellungen in einem Restaurant

Stellen Sie sich das so vor:

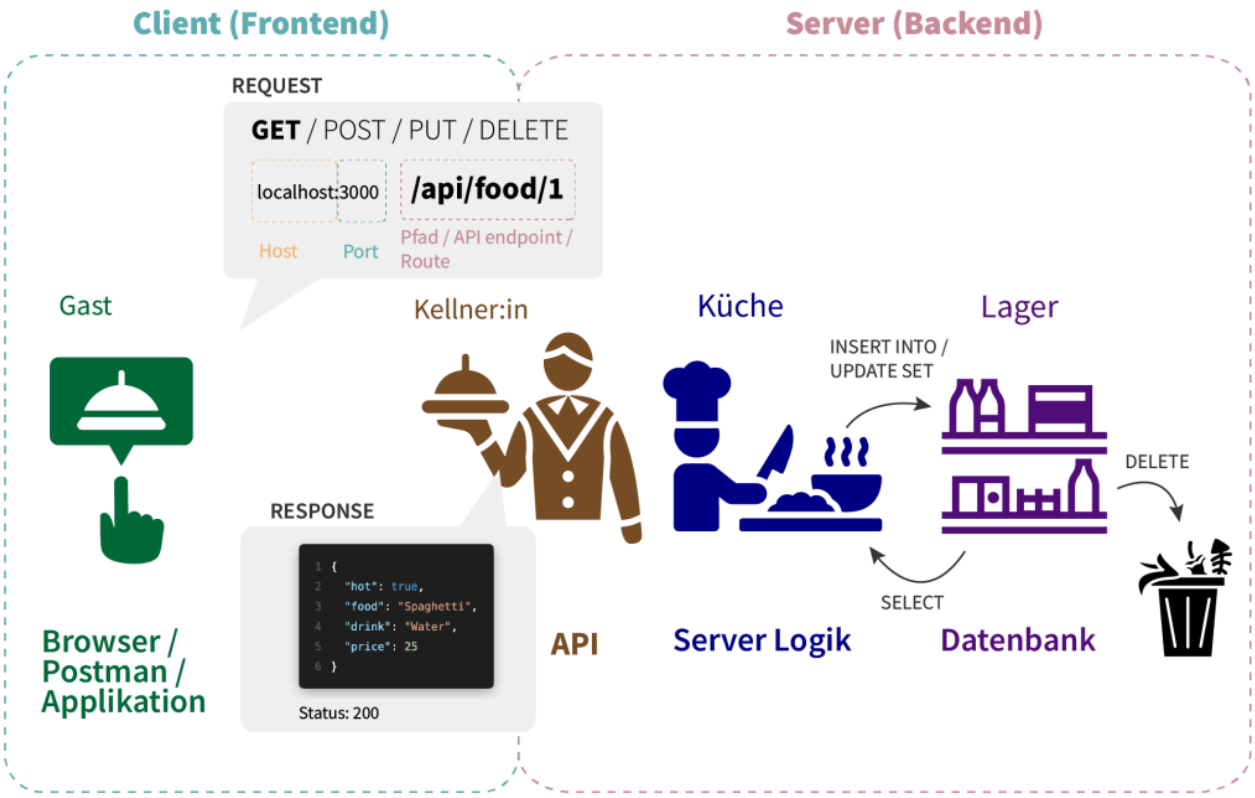
- **Client** (Browser oder Postman) = Gast im Restaurant [3)]
- **Server** (Express/Node.js) = Küche [4)]
- **HTTP-Anfrage** = Bestellung, die der Kellner/die Kellnerin zur Küche bringt
- **HTTP-Antwort** = fertiges Gericht, das der Kellner zurückbringt

Jede HTTP-Anfrage hat unter anderem:

- eine **Methode** (z.B. GET, POST, PUT, DELETE),
- einen **Pfad**[5)] (z.B. `/api/posts`, `/api/food/1`),
- optional einen **Body**[6)] (z.B. JSON bei POST/PUT)

Der Server reagiert auf eine Anfrage, indem er:

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



Route und Endpoint

Route (Express)

Eine **Route** ist die **Server-Funktion im Code**, die beschreibt, was bei einer bestimmten Kombination passiert:

- **HTTP-Methode + Pfad** (z.B. GET `/api/posts`)
- **Handler-Funktion**[7)] (z.B. Daten lesen, JSON senden)

```
(req, res) => {
    res.send('Hello World!');
}
```

Das hier ist die Handler-Funktion. `req` steht für Request (beinhaltet Anfrage-Parameter) und `res` steht für Response (Antwort-Objekt).

Beispiel: `app.get('/api/posts', …)` ist eine Route.

Endpoint (API)

Ein **API-Endpoint** ist eine Route, die als **Daten-Schnittstelle**[8] für Clients gedacht ist. Meistens liefert sie die Daten im **JSON**[9] Format zurück.

Beispiel: GET `/api/posts` → liefert Post-Daten als JSON (API-Endpoint).

CRUD & HTTP-Methoden

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

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

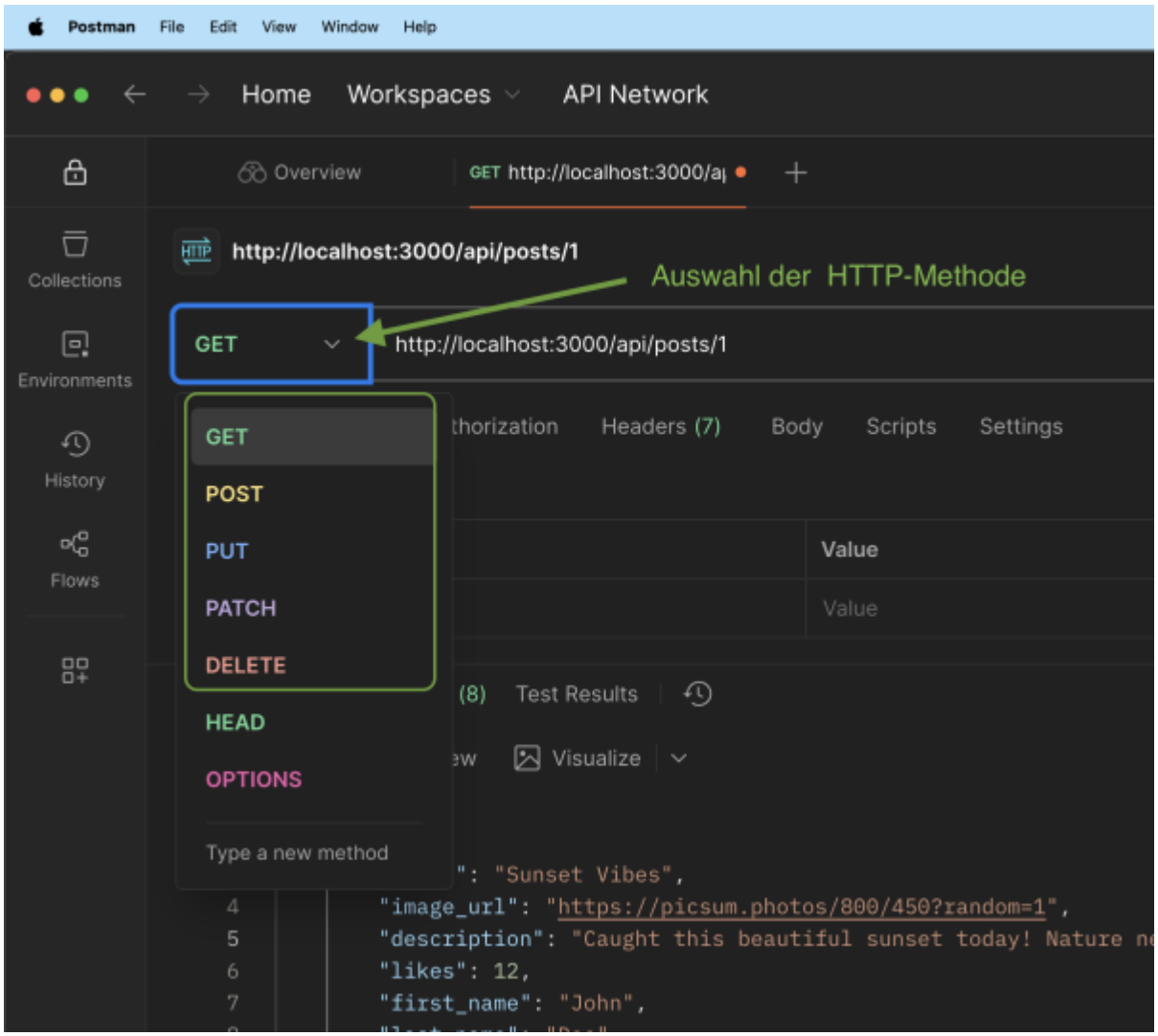
Später werden Sie für Ihre eigenen Use Cases (Reisen, Filme, Bücher, ...) genau solche Routen definieren, z.B. GET `/api/trips`, POST `/api/books`, usw.

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:
  - den **Statuscode** (z.B. 200, 201, 400, 404, 500) [10],
  - die **Response-Header** [11],
  - den **Response-Body** (Text oder JSON).
- Sie können im Body bequem JSON eingeben (z.B. neuen `post` anlegen).

Postman können Sie hier downloaden → Postman downloaden



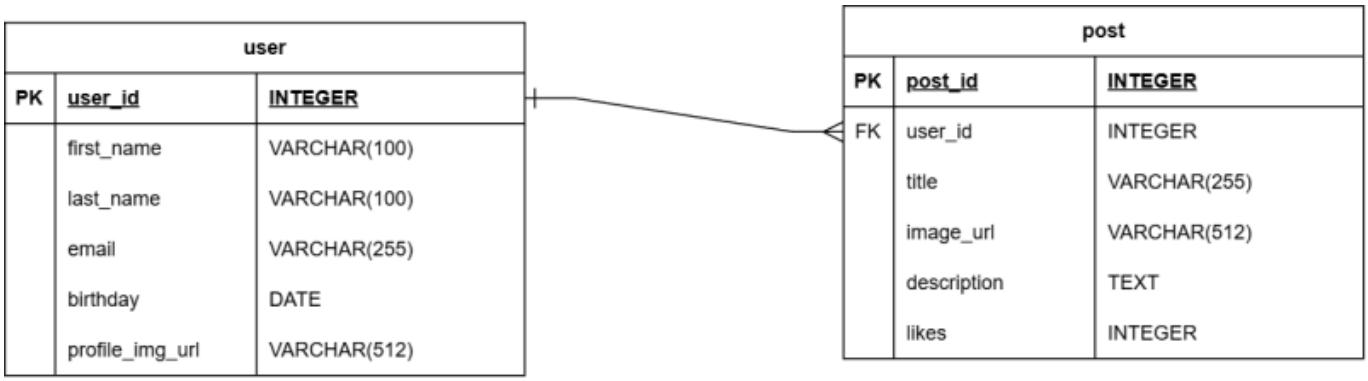
Screenshot Postman-Benutzeroberfläche mit den HTTP-Requests GET/POST/PUT/DELETE.

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

Beispiel-API: Social-Media-Posts

Wir verwenden als Beispiel eine vereinfachte `post`-Tabelle aus einer Social-Media-Datenbank:

- `post_id` (PK, Nummer)[12)]
- `user_id` (FK zum User – hier nur als Zahl)[13)]
- `title` (Titel des Posts)
- `image_url` (Bild-URL)
- `description` (Beschreibungstext)
- `likes` (Anzahl Likes)



Express-Server mit Endpoints erstellen

Wir erstellen einen Express-Server mit einer lokalen Liste von Posts (ohne Datenbank-Anbindung vorerst). Danach bauen wir die ersten Routen:

- GET `/api/posts` → alle Posts
- GET `/api/posts/:id` → ein Post anhand der ID
- POST `/api/posts` → neuen Post hinzufügen

Wie in Projekten aus der Realität arbeiten wir hier mit *api*-Routen (`/api/posts`) – das ist eine Konvention. Wir könnten auch nur `/posts` schreiben.

1. Start: Express-API für Posts

Wir bauen auf dem bekannten Server-Setup aus LU15 auf.

```
import express from 'express';

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

// Middleware: "sitzt" zwischen Anfrage und Route und
verarbeitet Daten.
// express.json() liest den Request-Body und macht daraus
```

```
req.body (JSON).
app.use(express.json());

// "Test-Tabelle" lokal
let posts = [
{
  post_id: 1,
  user_id: 1,
  title: 'Morning Coffee',
  image_url: 'https://picsum.photos/800/450?random=1',
  description: 'Nothing beats starting the day with a warm
cup of coffee.',
  likes: 10
},
{
  post_id: 2,
  user_id: 2,
  title: 'Sunset Vibes',
  image_url: 'https://picsum.photos/800/450?random=2',
  description: 'Caught this beautiful sunset today! Nature
never disappoints.',
  likes: 12
}
];

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

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

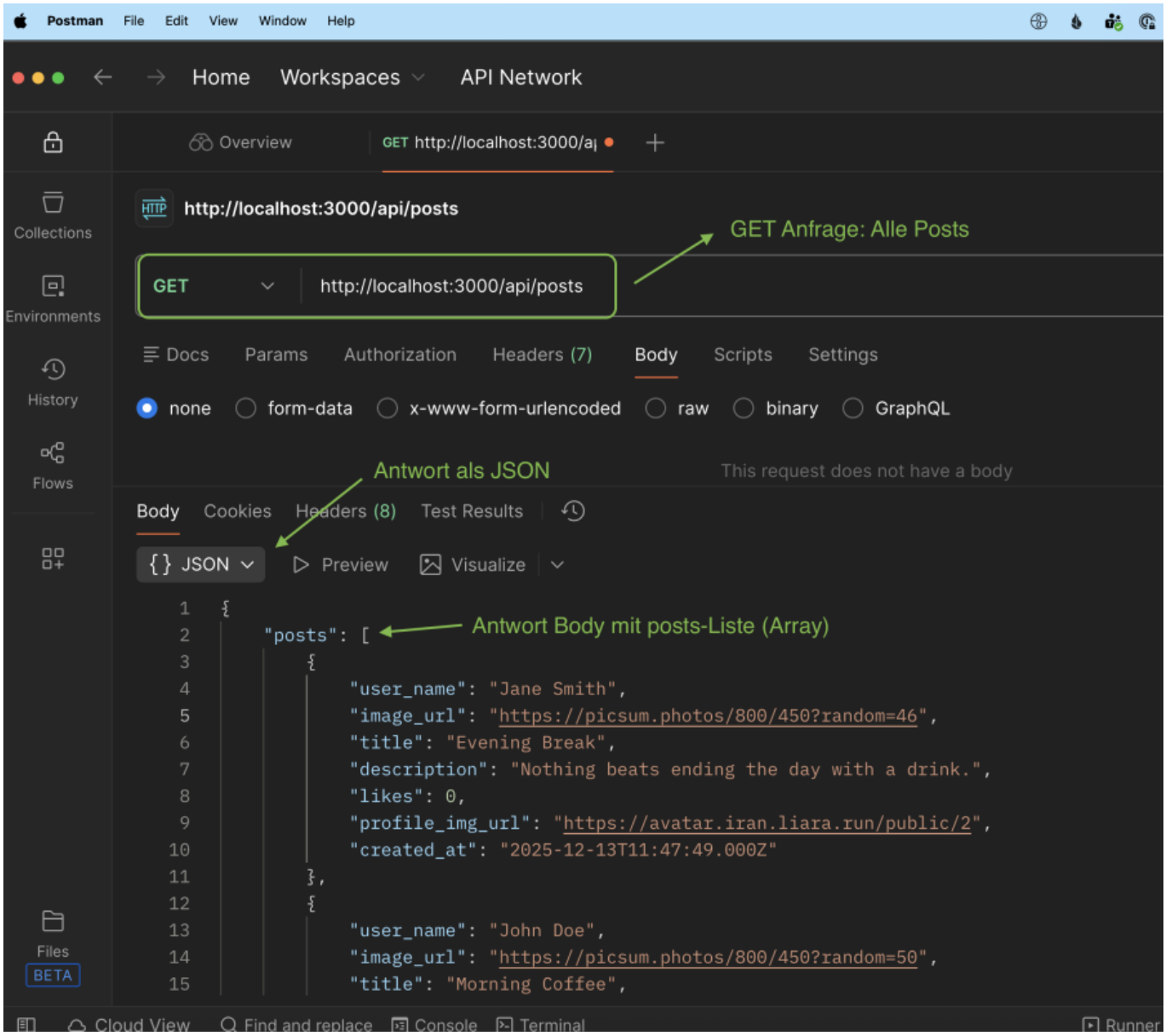
Wenn Sie diesen Server mit `npm run dev` starten (falls Nodemon → s. LU15 installiert, sonst `node index.js`), sollten Sie im Browser unter http://localhost:3000/ den Text **„API ist online“** sehen.

2. READ – Alle Posts abfragen

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

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

Test mit Postman oder Browser



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

3. READ – Einzelnen Post nach ID abfragen

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

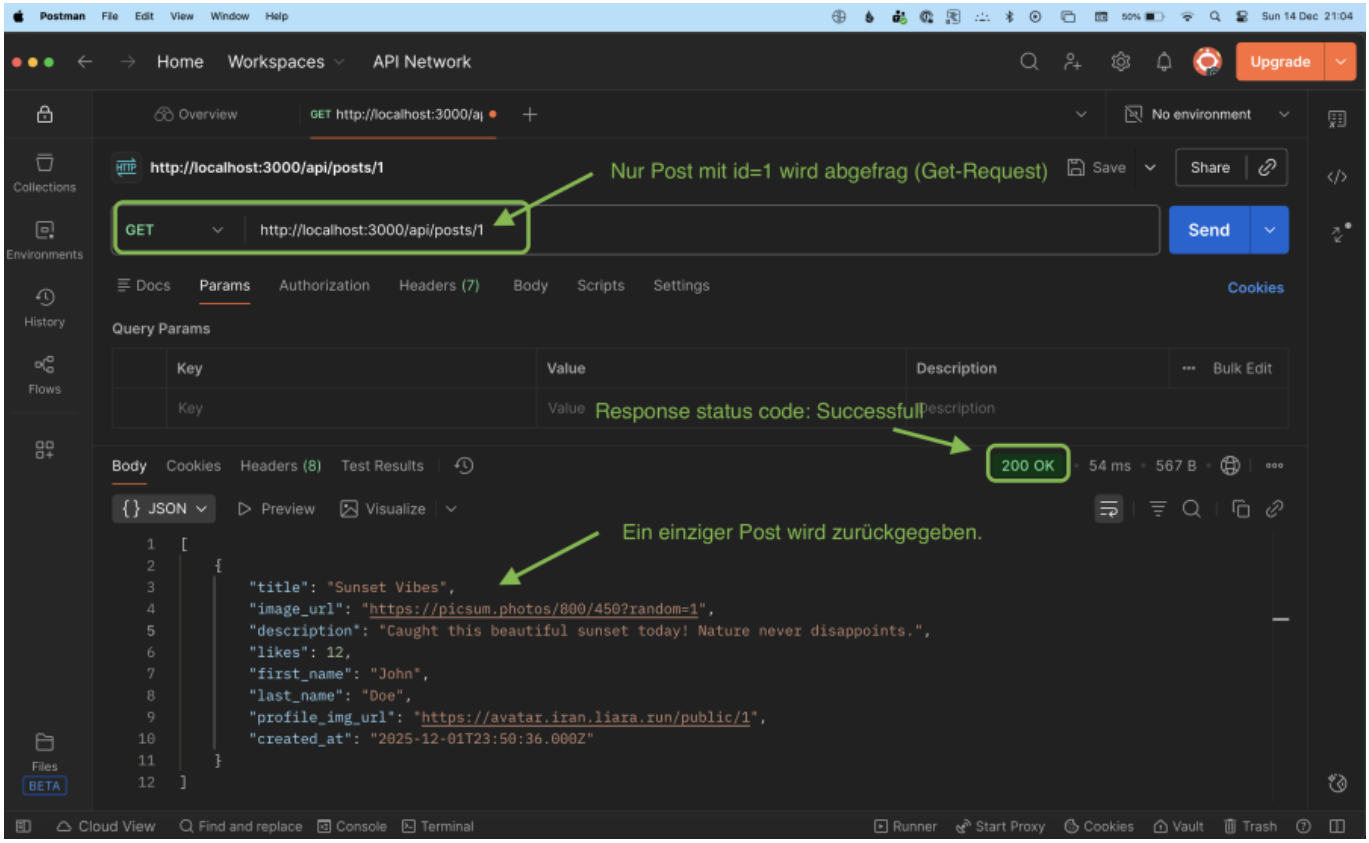
```
// READ – einzelner Post nach ID
app.get('/api/posts/:id', (req, res) => {
```

```
  const id = Number(req.params.id);      // Pfad-Parameter
(Route-Parameter) holen
  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/api/posts/1`
- Erwartung:
  - Status `200 OK`
  - JSON-Objekt mit `post_id: 1`
- URL: `http://localhost:3000/api/posts/999`
  - Status `404 Not Found`
  - Body: `Post nicht gefunden`

4. CREATE – Neuen Post anlegen

Nun soll ein neuer Post erstellt werden. Dazu schicken wir einen im Request einen „Body“ mit den

notwendigen Daten mit (im JSON-Format).

```
app.post('/api/posts', (req, res) => {
//Daten aus dem Anfrage-Objekt (Request) holen und in Variablen
speichern
const userId = req.body.user_id;
const title = req.body.title;
const imageUrl = req.body.image_url;
const description = req.body.description;

// ganz einfache Validierung (Pflichtfelder) - ohne titel oder
user_id (wer post erstellt hat) können wir keinen neuen Post
erstellen.
if (!title || !userId) {
    return res.status(400).send('Please enter a title and a
user_id');
}

// neue post_id berechnen - das wird in Zukunft die Datenbank
selbst machen.
// Letzte Post-ID aus dem Post-Array herausfinden:
const lastPostId = posts[posts.length - 1].post_id;
const newPostId = lastPostId + 1;

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

//neuer Post wird in Post-Array (Liste mit Posts) gespeichert
posts.push(newPost);

//Rückmeldung an Client: neuer Post und Status-Code 201
(Created)
res.status(201).json(newPost);


});
```

Erklärung

Pflichtfelder: `title` und `user_id` müssen vorhanden sein → sonst `400 Bad Request`.

Neue ID: Wir nehmen die `post_id` des letzten Elements im Array und zählen +1.

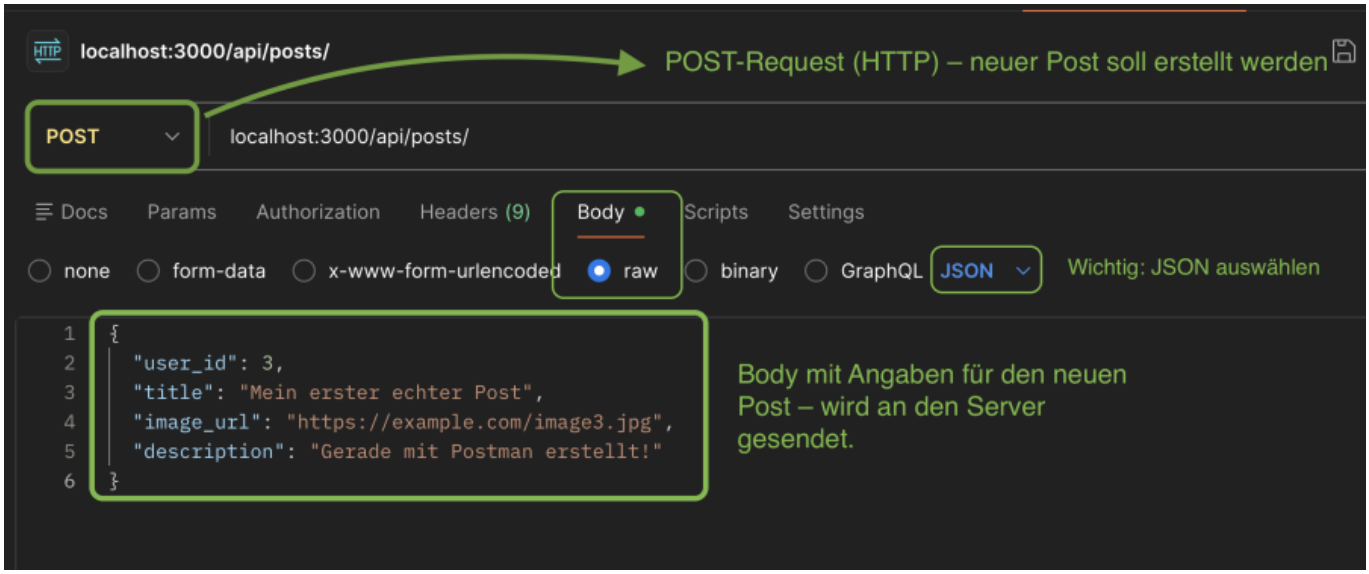
Antwort: `201 Created` + das neu erstellte Post-Objekt als JSON.

Dieser Ansatz mit `posts[posts.length - 1]` funktioniert nur, wenn:

- das Array mindestens 1 Element hat, und
- die Posts im Array nach `post_id` sortiert sind (letzter Post hat die höchste ID).

In echten Projekten übernimmt das später die Datenbank (AUTO_INCREMENT).

Test mit Postman



1. Methode: POST
2. URL: http://localhost:3000/api/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!"
}
```

1. Send klicken

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

Ausblick

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 lokalen JavaScript-Liste (Array) von `posts` erstellt (`GET /api/posts`, `GET /api/posts/:id`, `POST /api/posts`).

In der nächsten Unterrichtseinheit:

- ersetzen wir die JavaScript-Liste durch eine **MySQL-Tabelle `posts`**,
- 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.

[1] HTTP = „Hypertext Transfer Protocol“: ein Regelwerk, wie Clients und Server Daten austauschen.

[2] Backend-Server = Programm, das Anfragen entgegennimmt und Antworten zurückschickt; oft mit Datenbank-Logik dahinter.

[3] Client = Programm, das eine Anfrage sendet.

[4] Server = Programm, das Anfragen empfängt und Antworten liefert.

[5] Pfad = der Teil der URL nach Domain/Port, z.B. `/food/1`.

[6] Body = „Datenkörper“ der Anfrage, z.B. ein JSON-Objekt.

[7] Handler = Funktion, die ausgeführt wird, wenn die Route passt.

[8] Schnittstelle = klar definierter „Zugang“: welche URL, welche Methode, welche Daten kommen rein/raus.

[9] JSON = Datenformat, das wie ein JavaScript-Objekt aussieht; wird häufig in APIs verwendet.

[10] Statuscode = Zahl, die beschreibt, ob die Anfrage erfolgreich war und warum/nicht.

[11] Header = Zusatzinfos zur Antwort, z.B. Inhaltstyp oder Caching.

[12] PK = Primary Key / Primärschlüssel: eindeutige ID.

[13] FK = Foreign Key / Fremdschlüssel: verweist auf eine ID in einer anderen Tabelle.