

L17: RESTful APIs & SQL – Online-Shop Backend

Session 17 – Lecture

Dauer: 90 Minuten

Lernziele: LZ 2 (Relationale DB & SQL praktisch anwenden)

Block: 4 – Theorie, Optimierung & Polyglot

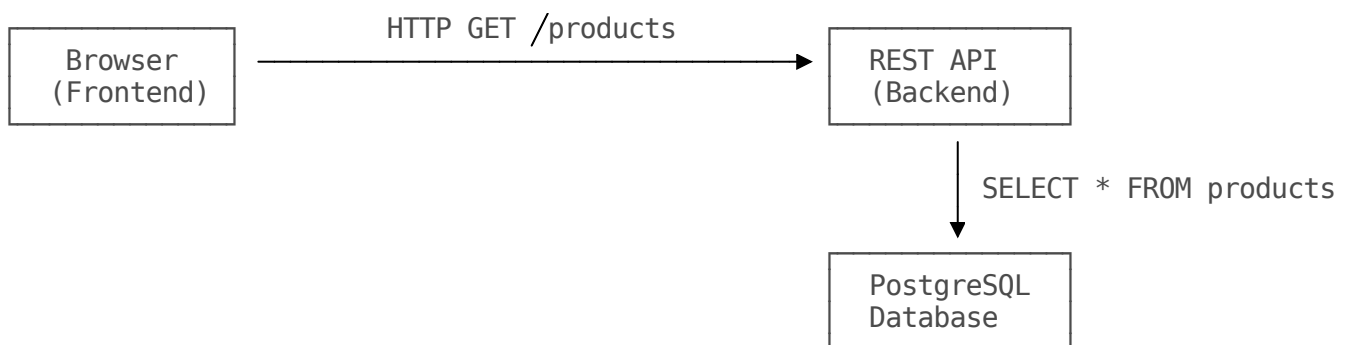
Willkommen zur siebzehnten Session! Heute verbinden wir zwei Welten: RESTful APIs und SQL. Sie lernen, wie moderne Web-APIs funktionieren und wie HTTP-Requests in SQL-Queries übersetzt werden. Das Besondere: Wir simulieren eine vollständige REST-API direkt im Browser – mit echter SQL-Ausführung in PGlite!

Stellen Sie sich vor: Sie bauen das Backend für einen Online-Shop. Frontend-Developer schicken HTTP-Requests an Ihre API, und Sie müssen diese in SQL-Queries übersetzen. Genau das üben wir heute – praxisnah und interaktiv!

Motivation: Warum REST & SQL?

Bevor wir loslegen, schauen wir uns an, wie RESTful APIs in der Praxis eingesetzt werden.





Szenario: Online-Shop Architecture



In echten Systemen übersetzt ein Backend-Server (z.B. Node.js, Python, Java) HTTP-Requests in SQL-Queries. Heute lernen Sie genau diese Übersetzung – und bauen sie selbst!

Live-Demo: Echte REST-API erkunden

Schauen wir uns zuerst eine echte öffentliche API an: die Fake Store API. Öffnen Sie die folgenden URLs in einem neuen Tab und beobachten Sie die JSON-Responses:

-  Alle Produkte: <https://fakestoreapi.com/products>
-  Ein Produkt: <https://fakestoreapi.com/products/1>
-  Kategorien: <https://fakestoreapi.com/products/categories>
-  Elektronik: <https://fakestoreapi.com/products/category/electronics>

Interaktive Demo: API im Browser testen

```

1  async function loadFakeStoreProducts() {
2    try {
3      const response = await fetch('https://fakestoreapi.com/products?limit=5');
4      const products = await response.json();
5
6      console.table(products);
7    } catch (error) {
8      console.error(error.message);
9    }
10 }
11
12 loadFakeStoreProducts();

```

[object Promise]

Unexpected token '<', "<!DOCTYPE "... is not valid JSON

Beeindruckend, oder? Diese Daten kommen von einem echten Server. Heute bauen Sie eine identische API – aber die Daten kommen aus Ihrer eigenen SQL-Datenbank im Browser!

Teil 1: Was ist eine REST-API?

REST steht für „Representational State Transfer“ – ein Architekturstil für Web-APIs. Klingt kompliziert? Ist es nicht! Im Kern geht es um vier einfache HTTP-Methoden.

HTTP-Methoden & CRUD

REST nutzt HTTP-Methoden, um Operationen auf Ressourcen auszuführen. Diese mappen direkt auf SQL-Operationen!

HTTP-Methode	Bedeutung	SQL-Operation	Beispiel-URL
GET	Daten abrufen (Read)	SELECT	GET /products
POST	Neue Daten erstellen (Create)	INSERT	POST /products
PUT	Daten aktualisieren (Update)	UPDATE	PUT /products/5
DELETE	Daten löschen (Delete)	DELETE	DELETE /products/5

Das Schöne: HTTP-Methoden und SQL-Operationen haben eine natürliche 1:1-Beziehung. GET wird zu SELECT, POST zu INSERT, DELETE zu DELETE!

URL-Struktur & Ressourcen

REST-APIs arbeiten mit Ressourcen, die über URLs identifiziert werden. Ressourcen entsprechen meist Datenbank-Tabellen.

URL-Pattern:

```
https://api.example.com/ressource
https://api.example.com/ressource/{id}
https://api.example.com/ressource?filter=value
```



Konkrete Beispiele:

URL	Beschreibung	SQL-Äquivalent
<code>GET /products</code>	Alle Produkte	<code>SELECT * FROM products</code>
<code>GET /products/5</code>	Produkt mit ID 5	<code>SELECT * FROM products WHERE product_id = 5</code>
<code>GET /products?category=Electronics</code>	Gefilterte Produkte	<code>SELECT * FROM products WHERE category = 'Electronics'</code>
<code>GET /customers/3/orders</code>	Bestellungen von Kunde 3	<code>SELECT * FROM orders WHERE customer_id = 3</code>

Sehen Sie das Muster? URLs beschreiben die Daten, die Sie wollen – und Sie übersetzen das in SQL!

HTTP-Status Codes

APIs kommunizieren Erfolg oder Fehler über HTTP-Status Codes. Die wichtigsten sollten Sie kennen.

Status Code	Bedeutung	Wann verwenden?
200 OK	Erfolg	Daten erfolgreich abgerufen/geändert
201 Created	Ressource erstellt	Nach erfolgreichem INSERT
400 Bad Request	Ungültige Anfrage	Fehlende/falsche Parameter
404 Not Found	Ressource nicht gefunden	Keine Daten in Datenbank
500 Internal Server Error	Server-Fehler	SQL-Fehler, Constraint-Verletzung

Diese Codes helfen dem Frontend zu verstehen, was passiert ist – ohne die Response-Daten zu parsen!

JSON als Datenformat

REST-APIs senden und empfangen Daten im JSON-Format. JSON ist leichtgewichtig und JavaScript-nativ.

Response-Beispiel:

```
{
  "data": [
    {
      "product_id": 1,
      "product_name": "Laptop",
      "price": 999.99
    },
    {
      "product_id": 2,
      "product_name": "Mouse",
      "price": 29.99
    }
  ],
  "status": "success",
  "count": 2
}
```

Ihre SQL-Query-Ergebnisse werden in dieses Format konvertiert – automatisch durch die API-Schicht!

Teil 2: HTTP → SQL Mapping

Jetzt wird es praktisch! Wir schauen uns an, wie jede HTTP-Operation in eine SQL-Query übersetzt wird.

GET → SELECT

GET-Requests rufen Daten ab – das einfachste Mapping.

Pattern 1: Alle Ressourcen

```
GET /products
↓
SELECT * FROM products ORDER BY product_name;
```

Pattern 2: Eine Ressource nach ID

```
GET /products/5
↓
SELECT * FROM products WHERE product_id = 5;
```

Pattern 3: Gefilterte Ressourcen

```
GET /products?category=Electronics
↓
SELECT p.*
FROM products p
```

```
INNER JOIN product_categories pc ON p.product_id = pc.product_id
INNER JOIN categories c ON pc.category_id = c.category_id
WHERE c.category_name = 'Electronics';
```

Pattern 4: Verschachtelte Ressourcen (Joins)

```
GET /customers/3/orders
↓
SELECT o.*
FROM orders o
WHERE o.customer_id = 3
ORDER BY o.order_date DESC;
```

Sehen Sie das Muster? URL-Parameter werden zu WHERE-Bedingungen, verschachtelte Pfade zu Joins!

POST → INSERT

POST-Requests erstellen neue Daten. Der Request-Body enthält die Werte.

Request:

```
POST /products
Content-Type: application/json

{
  "product_name": "Webcam",
  "price": 89.99
}
```

SQL-Translation:

```
INSERT INTO products (product_name, price)
VALUES ('Webcam', 89.99)
RETURNING product_id, product_name, price;
```

Wichtig: RETURNING gibt die neu erstellte Zeile zurück – inklusive auto-generierter ID! Das ist das Ergebnis der POST-Response.

Response:

```
{
  "data": {
    "product_id": 10,
    "product_name": "Webcam",
    "price": 89.99
  },
  "status": "success",
  "message": "Product created"
}
```

DELETE → DELETE

DELETE-Requests entfernen Daten.

Request:

```
DELETE /products/7
```



SQL-Translation:

```
DELETE FROM products WHERE product_id = 7;
```



Response:

```
{
  "status": "success",
  "message": "Product deleted",
  "deleted_id": 7
}
```



Achtung: Was passiert, wenn das Produkt in order_items referenziert wird? Foreign Key Constraint! Das ist ein 500-Fehler – dazu später mehr.

PUT/PATCH → UPDATE (Optional)

PUT aktualisiert eine komplette Ressource, PATCH nur Teile davon. Heute fokussieren wir auf POST und DELETE, aber hier das Konzept:

```
PUT /products/5
Content-Type: application/json
```



```
{
  "product_name": "Gaming Mouse Pro",
  "price": 39.99
}
```

```
UPDATE products
SET product_name = 'Gaming Mouse Pro',
    price = 39.99
WHERE product_id = 5
RETURNING *;
```



Datenbank-Setup: Online-Shop

Bevor wir mit der API-Implementierung starten, initialisieren wir unsere Datenbank. Wir nutzen das bekannte E-Commerce-Schema.



```
1  -- Locations
2  CREATE TABLE locations (
3      location_id INTEGER PRIMARY KEY,
4      city TEXT NOT NULL,
5      postal_code TEXT NOT NULL,
6      country TEXT DEFAULT 'Germany'
7  );
8
9  -- Categories
10 CREATE TABLE categories (
11     category_id INTEGER PRIMARY KEY,
12     category_name TEXT NOT NULL UNIQUE,
13     description TEXT
14 );
15
16 -- Customers
17 CREATE TABLE customers (
18     customer_id INTEGER PRIMARY KEY,
19     first_name TEXT NOT NULL,
20     last_name TEXT NOT NULL,
21     email TEXT UNIQUE,
22     street TEXT,
23     street_number TEXT,
24     location_id INTEGER REFERENCES locations(location_id)
25 );
26
27 -- Orders
28 CREATE TABLE orders (
29     order_id INTEGER PRIMARY KEY,
30     customer_id INTEGER REFERENCES customers(customer_id),
31     order_date DATE,
32     total_amount DECIMAL(10,2),
33     status TEXT
34 );
35
36 -- Products
37 CREATE TABLE products (
38     product_id SERIAL PRIMARY KEY,
39     product_name TEXT NOT NULL,
40     price DECIMAL(10,2)
41 );
42
43 -- Product_Categories
44 CREATE TABLE product_categories (
45     product_id INTEGER REFERENCES products(product_id),
46     category_id INTEGER REFERENCES categories(category_id),
47     PRIMARY KEY (product_id, category_id)
48 );
49
50 -- Order_Items
```

```
51 CREATE TABLE order_items (  
52     order_item_id INTEGER PRIMARY KEY,  
53     order_id INTEGER REFERENCES orders(order_id),  
54     product_id INTEGER REFERENCES products(product_id),  
55     quantity INTEGER,  
56     line_total DECIMAL(10,2)  
57 );  
58  
59 -- Sample Data  
60 INSERT INTO locations VALUES (1, 'Berlin', '10115', 'Germany'), (2,  
    'Hamburg', '20095', 'Germany');  
61 INSERT INTO categories VALUES (1, 'Electronics', 'Electronic devices'  
    'Furniture', 'Office furniture');  
62 INSERT INTO customers VALUES  
63     (1, 'Alice', 'Smith', 'alice@example.com', 'Main St', '42', 1),  
64     (2, 'Bob', 'Johnson', 'bob@example.com', 'Oak Ave', '15', 2);  
65 INSERT INTO products (product_name, price) VALUES  
66     ('Laptop', 999.99), ('Mouse', 29.99), ('Keyboard', 79.99),  
67     ('Monitor', 299.99), ('Desk Chair', 199.99);  
68 INSERT INTO product_categories VALUES (1,1), (2,1), (3,1), (4,1), (5,  
69 INSERT INTO orders VALUES (101, 1, '2024-01-15', 299.99, 'delivered')  
    , 2, '2024-01-22', 999.99, 'delivered');  
70 INSERT INTO order_items VALUES (1, 101, 4, 1, 299.99), (2, 102, 1, 1,  
    .99);
```

```
-- Locations
CREATE TABLE locations (
  location_id INTEGER PRIMARY KEY,
  city TEXT NOT NULL,
  postal_code TEXT NOT NULL,
  country TEXT DEFAULT 'Germany'
)
```

ok

```
-- Categories
CREATE TABLE categories (
  category_id INTEGER PRIMARY KEY,
  category_name TEXT NOT NULL UNIQUE,
  description TEXT
)
```

ok

```
-- Customers
CREATE TABLE customers (
  customer_id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT UNIQUE,
  street TEXT,
  street_number TEXT,
  location_id INTEGER REFERENCES locations(location_id)
)
```

ok

```
-- Orders
CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER REFERENCES customers(customer_id),
  order_date DATE,
  total_amount DECIMAL(10,2),
  status TEXT
)
```

ok

```
-- Products
CREATE TABLE products (
  product_id SERIAL PRIMARY KEY,
  product_name TEXT NOT NULL,
  price DECIMAL(10,2)
)
```

ok

-- Product_Categories

```
CREATE TABLE product_categories (  
  product_id INTEGER REFERENCES products(product_id),  
  category_id INTEGER REFERENCES categories(category_id),  
  PRIMARY KEY (product_id, category_id)  
)
```

ok

-- Order_Items

```
CREATE TABLE order_items (  
  order_item_id INTEGER PRIMARY KEY,  
  order_id INTEGER REFERENCES orders(order_id),  
  product_id INTEGER REFERENCES products(product_id),  
  quantity INTEGER,  
  line_total DECIMAL(10,2)  
)
```

ok

-- Sample Data

```
INSERT INTO locations VALUES (1, 'Berlin', '10115', 'Germany'), (2, 'Hamburg',  
'20095', 'Germany')
```

ok

```
INSERT INTO categories VALUES (1, 'Electronics', 'Electronic devices'), (2, 'Furniture',  
'Office furniture')
```

ok

INSERT INTO customers VALUES

```
(1, 'Alice', 'Smith', 'alice@example.com', 'Main St', '42', 1),  
(2, 'Bob', 'Johnson', 'bob@example.com', 'Oak Ave', '15', 2)
```

ok

INSERT INTO products (product_name, price) VALUES

```
('Laptop', 999.99), ('Mouse', 29.99), ('Keyboard', 79.99),  
( 'Monitor', 299.99), ('Desk Chair', 199.99)
```

ok

```
INSERT INTO product_categories VALUES (1,1), (2,1), (3,1), (4,1), (5,2)
```

ok

```
INSERT INTO orders VALUES (101, 1, '2024-01-15', 299.99, 'delivered'), (102, 2, '2024-01-22', 999.99, 'delivered')
```

ok

```
INSERT INTO order_items VALUES (1, 101, 4, 1, 299.99), (2, 102, 1, 1, 999.99)
```

ok

Perfekt! Unsere Datenbank ist bereit. Jetzt implementieren wir die API-Schicht!

Playground



1 ERDIAGRAM

```
https://dbdiagram.io/embed?c=VGFiBGUgY2F0ZWdvcmlscyB7CiAgY2F0ZWdvcnlfYWQgaW50IFtwaywgbm90IG51bGxdCiAg
```

Teil 3: API-Setup – fetch-Override

Hier kommt die Magie: Wir überschreiben die globale `fetch`-Funktion, um HTTP-Requests abzufangen und in SQL-Queries zu übersetzen!

Architektur:

JavaScript Code

Browser Database

```
fetch('http://little-amazon.com/products')
├──> URL-Prüfung: little-amazon.com?
├──> JA → Route zu SQL mappen
│   ├──> SQL in PGLite ausführen
│   └──> JSON-Response zurückgeben
```

Implementation (vorgegeben):

```
1 // ===== Einfacher Router (inspiriert von Express.js) =====
2 class SimpleRouter {
3   constructor() {
4     this.routes = { GET: {}, POST: {}, DELETE: {} };
5     this._requestBody = null;
6   }
7
8   get(path, handler) {
9     this.routes.GET[path] = { pattern: this._pathToRegex(path), handler: handler };
10  }
11
12  post(path, handler) {
13    this.routes.POST[path] = { pattern: this._pathToRegex(path), handler: handler };
14  }
15
16  delete(path, handler) {
17    this.routes.DELETE[path] = { pattern: this._pathToRegex(path), handler: handler };
18  }
19
20  _pathToRegex(path) {
21    // Konvertiert /products/:id zu Regex mit Named Groups
22    const paramNames = [];
23    const regexPattern = path
24      .replace(/:\w+/g, (match) => {
25        paramNames.push(match.slice(1)); // ':id' -> 'id'
26        return '([^/]+)'; // Match alles außer /
27      })
28      .replace(/\\/g, '\\\\'); // Escape /
29
30    return { regex: new RegExp(`^${regexPattern}$`), paramNames };
31  }
32
33  async handle(method, path, body) {
34    // Body für POST-Handler verfügbar machen
35    if (body) {
36      try {
37        this._requestBody = JSON.parse(body);
38      } catch (e) {
39        this._requestBody = body;
40      }
41    }
42
43    const routes = this.routes[method] || {};
44
45    for (const [routePath, route] of Object.entries(routes)) {
46      const match = path.match(route.pattern.regex);
47      if (match) {
```

```
48 // Extrahiere Parameter (z.B. { id: '5' })
49 const params = {};
50 route.pattern.paramNames.forEach((name, i) => {
51     params[name] = match[i + 1];
52 });
53
54 try {
55     return await route.handler(params, this._requestBody);
56 } catch (error) {
57     return {
58         status: 'error',
59         message: `Handler error: ${error.message}`,
60         httpStatus: 500
61     };
62 }
63 }
64 }
65
66 return { status: 'error', message: 'Endpoint not found', httpStat
67     404 };
68 }
69
70 // Router-Instanz erstellen
71 const router = new SimpleRouter();
72 window.router = router;
```

```
1 // ===== Globaler fetch-Override =====
2 window.originalFetch = window.fetch;
3
4 window.fetch = async function(url, options = {}) {
5   // Nur little-amazon.com abfangen
6   if (typeof url === 'string' && url.startsWith('http://little-amazon
7   )) {
8     const path = url.replace('http://little-amazon.com', '');
9     const method = options.method || 'GET';
10
11     try {
12       const result = await router.handle(method, path, options.body);
13       return new Response(JSON.stringify(result), {
14         status: result.httpStatus || 200,
15         headers: { 'Content-Type': 'application/json' }
16       });
17     } catch (error) {
18       return new Response(JSON.stringify({
19         status: 'error',
20         message: error.message
21       }), {
22         status: 500,
23         headers: { 'Content-Type': 'application/json' }
24       });
25     }
26
27     // Normale Requests durchreichen
28     return window.originalFetch(url, options);
29   };
30
31   // ===== Routen definieren (TODO: Implementieren Sie die Handler
32   // =====
33   // Syntax-Beispiele (noch nicht implementiert):
34   //
35   // router.get('/products', async (params) => {
36   //   // Handler für GET /products
37   //   return { status: 'success', data: [...], httpStatus: 200 };
38   // });
39   //
40   // router.get('/products/:id', async (params) => {
41   //   const productId = params.id; // ✨ Parameter automatisch extrah
42   //   // Handler für GET /products/:id
43   // });
44   //
45   // router.post('/products', async (params) => {
46   //   const data = router._requestBody; // Body als JSON-Objekt
```

```

47 // // Handler für POST /products
48 // });
49 //
50 // router.delete('/products/:id', async (params) => {
51 //   const productId = params.id;
52 //   // Handler für DELETE /products/:id
53 // });
54
55 console.log('✅ Little Amazon API mit Router loaded!');
56 console.log('📖 Routen-Syntax: router.get("/products/:id", async (par
    => { ... })');
57 console.log('ℹ️ Response-Format: { status: "success"|"error", data:
    httpStatus: 200|404|500 }');

```

```

✅ Little Amazon API mit Router loaded!
📖 Routen-Syntax: router.get("/products/:id", async (params) => { ...
})
ℹ️ Response-Format: { status: "success"|"error", data: [...],
httpStatus: 200|404|500 }
undefined

```

Perfekt! Jetzt haben wir einen eleganten Router! Statt verschachtelter if/else nutzen Sie

`router.get('/products/:id', handler)` – genau wie in Express.js oder Next.js!

Teil 4: Hands-on – SELECT Queries

Beginnen wir mit GET-Requests. Ihre Aufgabe: Schreiben Sie SQL-Queries für verschiedene API-Endpunkte!

Aufgabe 1: Alle Produkte laden

Implementieren Sie `GET /products` – zeigen Sie alle Produkte an.

TODO: Ersetzen Sie den Handler mit Ihrer SQL-Query

```

1 // TODO: Implementieren Sie den Handler für GET /products
2 router.get('/products', async (params) => {
3   // Ihre SQL-Query hier:
4   const query = `
5     -- SELECT alle Produkte, sortiert nach product_name
6
7   `;
8
9   const result = await db.query(query);
10  return {
11    status: 'success',
12    data: result.rows,
13    count: result.rows.length,
14    httpStatus: 200
15  };
16 }

```

```
17  
18 console.debug('✅ GET /products implementiert');
```

✅ GET /products implementiert

Test: Rufen Sie die API auf!

```
1 try {  
2   const response = await fetch('http://little-amazon.com/products');  
3   const data = await response.json();  
4  
5   if (data.status === 'error') {  
6     throw new Error(data.message || 'Unknown error');  
7   }  
8  
9   console.table(data.data);  
10 } catch (error) {  
11   console.error(error.message);  
12 }
```

[]

Sobald Ihre Query funktioniert, sehen Sie die Produktliste! Falls „not implemented“ erscheint, fehlt noch die SQL-Query.

Aufgabe 2: Produkt nach ID

Implementieren Sie `GET /products/{id}` – zeigen Sie ein einzelnes Produkt.

TODO: Erweitern Sie `handleGET()` um ID-Routing

```
1 router.get('/products/:id', async (params) => {  
2   const productId = params.id;  
3  
4   const query = `  
5     -- TODO: product_id = ${productId}  
6   `;  
7  
8  
9   const result = await db.query(query);  
10  
11   if (result.rows.length === 0) {  
12     return { status: 'error', message: 'Product not found', httpStatus: 404 };  
13   }  
14  
15   return { status: 'success', data: result.rows, httpStatus: 200 };  
16 });  
17
```

```
18 console.debug('✅ GET /products/:id implementiert');
```

```
✅ GET /products/:id implementiert
```

```
1 try {
2   const response = await fetch('http://little-amazon.com/products/2')
3   const data = await response.json();
4
5   if (data.status === 'error') {
6     throw new Error(data.message || 'Unknown error');
7   }
8
9   console.table(data.data);
10 } catch (error) {
11   console.error(error.message);
12 }
```

```
Product not found
```

Aufgabe 3: Produkte einer Kategorie (JOIN)

Implementieren Sie `GET /products/category/{name}` – nutzen Sie einen JOIN!

TODO: Erweitern Sie `handleGET()` um Kategorie-Filter

```
1 router.get('/products/category/:name', async (params) => {
2   const categoryName = decodeURIComponent(params.name);
3
4   const query = `
5     -- TODO: SELECT mit JOIN über product_categories und categories
6     -- WHERE c.category_name = '${categoryName}'
7
8   `;
9
10  const result = await db.query(query);
11  return { status: 'success', data: result.rows, count: result.rows.length };
12 });
13
14 console.debug('✅ GET /products/category/{name} implementiert');
```

```
✅ GET /products/category/{name} implementiert
```

```
1 try {
2   const response = await fetch('http://little-amazon.com/products/cat
```

```

    /Electronics');
3   const data = await response.json();
4
5   if (data.status === 'error') {
6       throw new Error(data.message || 'Unknown error');
7   }
8
9   console.table(data.data);
10  } catch (error) {
11      console.error(error.message);
12  }

```

[]

Aufgabe 4: Kunden mit Bestellungen (JOIN + Aggregation)

Implementieren Sie `GET /customers/{id}/orders` – zeigen Sie alle Bestellungen eines Kunden.

```

1  router.get('/customers', async () => {
2      const query = `
3          -- TODO: SELECT alle Kunden
4      `;
5
6      const result = await db.query(query);
7      return { status: 'success', data: result.rows, count: result.rows.length };
8  });
9
10
11 console.debug('✅ GET /customers implementiert');
12
13 router.get('/customers/:id/orders', async (params) => {
14     const customerId = params.id;
15
16     const query = `
17         -- WHERE c.customer_id = '${customerId}'
18     `;
19
20
21     const result = await db.query(query);
22     return { status: 'success', data: result.rows, count: result.rows.length };
23 });
24
25 console.debug('✅ GET /customers/:id/orders implementiert');

```

✅ GET /customers implementiert
 ✅ GET /customers/:id/orders implementiert

```

1 try {
2   const response = await fetch('http://little-amazon.com/customers/1/orders');
3   const data = await response.json();
4
5   if (data.status === 'error') {
6     throw new Error(data.message || 'Unknown error');
7   }
8
9   console.table(data.data);
10 } catch (error) {
11   console.error(error.message);
12 }

```

[]

Teil 5: Hands-on – INSERT Queries

Jetzt wird es spannend: POST-Requests erstellen neue Daten! Der Request-Body enthält die Werte als JSON.

Aufgabe 5: Neues Produkt hinzufügen

Implementieren Sie `POST /products` – erstellen Sie ein neues Produkt.

TODO: Implementieren Sie POST-Handler mit Body-Parsing

```

1 // TODO: Implementieren Sie POST /products
2 router.post('/products', async (params) => {
3   const data = router._requestBody || {};
4
5   // Validierung
6   if (!data.product_name || !data.price) {
7     return {
8       status: 'error',
9       message: 'Missing required fields: product_name, price',
10      httpStatus: 400
11    };
12  }
13
14  if (data.price < 0) {
15    return {
16      status: 'error',
17      message: 'Price must be positive',
18      httpStatus: 400
19    };
20  }
21
22  // TODO: INSERT-Query mit RETURNING

```

```

23   const query = `
24       -- Ihre INSERT-Query hier
25
26       RETURNING product_id, product_name, price;
27   `;
28
29   const result = await db.query(query);
30   return {
31       status: 'success',
32       message: 'Product created',
33       data: result.rows[0],
34       httpStatus: 201
35   };
36   });
37
38   console.debug('✅ POST /products implementiert');

```

✅ POST /products implementiert

Test: Produkt erstellen

Produktname:

Preis (€):

 Produkt erstellen

Hinweis: In echten Systemen würden Sie Prepared Statements nutzen, um SQL-Injection zu verhindern!

</details>

Aufgabe 6: Produkte löschen (DELETE)

Implementieren Sie `POST /customers` – erstellen Sie einen neuen Kunden.

DELETE-Handler

```

1  // TODO: Implementieren Sie DELETE /products/:id
2  router.delete('/products/:id', async (params) => {
3      const productId = params.id;
4
5      // Prüfen ob Produkt existiert
6      const checkQuery = `SELECT product_id FROM products WHERE product_i
7                          ${productId}`;
8      const checkResult = await db.query(checkQuery);

```

```

8
9 ▾ if (checkResult.rows.length === 0) {
10     return { status: 'error', message: 'Product not found', httpStatus: 404 };
11 }
12
13 ▾ try {
14     const deleteProductQuery = `
15         -- Ihre DELETE-Query für products hier
16     `;
17
18     await db.query(deleteProductQuery);
19
20     return {
21         status: 'success',
22         message: 'Product deleted',
23         deleted_id: parseInt(productId),
24         httpStatus: 200
25     };
26 } catch (error) {
27     // Foreign Key Constraint Fehler abfangen
28     if (error.message.includes('foreign key constraint')) {
29         return {
30             status: 'error',
31             message: 'Cannot delete product: still referenced in other tables',
32             detail: error.message,
33             httpStatus: 409 // Conflict
34         };
35     }
36     throw error;
37 }
38 }
39 });
40
41 console.debug('✅ DELETE /products/:id implementiert');

```

✅ DELETE /products/:id implementiert

Test: Mini-Shop mit Löschen-Funktion

 Shop laden

► 💡 Wichtig

Aufgabe 8: Kunde löschen (CASCADE-Problem)

Was passiert, wenn Sie einen Kunden löschen, der Bestellungen hat? Foreign Key Constraint! Genau wie beim Produkt-Löschen.

Probieren Sie es aus:

```
1 router.delete('/customers/:id', async (params) => {
2   const customerId = params.id;
3
4   // Prüfen ob Kunde existiert
5   const checkQuery = `SELECT customer_id FROM customers WHERE customer_id = ${customerId}`;
6   const checkResult = await db.query(checkQuery);
7
8   if (checkResult.rows.length === 0) {
9     return { status: 'error', message: 'Customer not found', httpStatus: 404 };
10  }
11
12  try {
13    const result = await db.query(`DELETE FROM customers WHERE customer_id = ${customerId}`);
14    return {
15      status: 'success',
16      message: 'Customer deleted',
17      deleted_id: parseInt(customerId),
18      httpStatus: 200
19    };
20  } catch (error) {
21    // Foreign Key Constraint Fehler abfangen
22    if (error.message.includes('foreign key constraint')) {
23      return {
24        status: 'error',
25        message: 'Cannot delete customer: still referenced in other tables',
26        detail: error.message,
27        httpStatus: 409 // Conflict
28      };
29    }
30    throw error;
31  }
32 });
```

Unexpected token '.'

```
1 // Versuchen Sie Kunde 1 zu löschen (hat Bestellungen!)
2 const response = await fetch('http://little-amazon.com/customers/1', {
3   method: 'DELETE'
4 });
5 const data = await response.json();
```

```
6 console.log(data);
```

```
{"status":"error","message":"Endpoint not found","httpStatus":404}
```

Sie bekommen einen Fehler! Die Datenbank verhindert das Löschen wegen der Foreign Key Referenzen in der `orders`-Tabelle. Das ist gewollt – Datenkonsistenz!

Lösungen für Foreign Key Probleme:

1. **Reihenfolge beachten:** Erst abhängige Daten löschen, dann Hauptdaten `\`sql DELETE FROM productcategories WHERE productid = 5; DELETE FROM products WHERE product_id = 5; \``
2. **Soft Delete:** Setzen Sie `deleted = true` statt echtem DELETE `\`sql UPDATE products SET deleted = true WHERE product_id = 5; \``
3. **CASCADE:** `ON DELETE CASCADE` in der Tabellendefinition `\`sql CREATE TABLE product_categories (productid INTEGER REFERENCES products(productid) ON DELETE CASCADE, ...); \``
4. **Transaktionen:** Mehrere Deletes atomar ausführen `\`sql BEGIN; DELETE FROM orderitems WHERE productid = 5; DELETE FROM productcategories WHERE productid = 5; DELETE FROM products WHERE product_id = 5; COMMIT; \``

Teil 7: Error-Handling

Fehler gehören zur Realität! Lassen Sie uns verschiedene Fehlertypen simulieren und richtig behandeln.

404: Ressource nicht gefunden

Wenn eine Query keine Daten zurückgibt, sollten Sie 404 zurückgeben.

```
1 // Beispiel: GET /products/9999
2 const result = await db.query('SELECT * FROM products WHERE product_i
  9999');
3
4 if (result.rows.length === 0) {
5   return {
6     status: 'error',
7     message: 'Product not found',
8     httpStatus: 404
9   };
10 }
```

400: Ungültige Daten

Validieren Sie Inputs, bevor Sie SQL ausführen!

```
// Beispiel: Negativer Preis
```

```

if (data.price < 0) {
  return {
    status: 'error',
    message: 'Price must be positive',
    httpStatus: 400
  };
}

// Beispiel: Fehlende Pflichtfelder
if (!data.product_name || !data.price) {
  return {
    status: 'error',
    message: 'Missing required fields: product_name, price',
    httpStatus: 400
  };
}

```

500: SQL-Fehler

Foreign Key Constraints, Syntax-Fehler, etc. führen zu 500-Fehlern.

Beispiel: Foreign Key Constraint

```

try {
  const result = await db.query(`SELECT * FROM product_list`);
  return { status: 'success', data: result.rows };
} catch (error) {
  console.error('SQL Error:', error);

  // Spezifische Fehlerbehandlung
  if (error.message.includes('foreign key constraint')) {
    return {
      status: 'error',
      message: 'Cannot delete: record is still referenced by other tables',
      detail: error.message,
      httpStatus: 409 // Conflict
    };
  }

  return {
    status: 'error',
    message: `Database error: ${error.message}`,
    httpStatus: 500
  };
}

```

HTTP 409 (Conflict) ist der richtige Code für Foreign Key Constraint Fehler – es ist kein Server-Fehler, sondern ein Konflikt mit der Datenintegrität!

Wrap-up & Best Practices

Fassen wir zusammen, was Sie heute gelernt haben!

HTTP → SQL Mapping:

- ✓ GET → SELECT (mit WHERE für Filter, JOIN für Relations)
- ✓ POST → INSERT (mit RETURNING für Response)
- ✓ DELETE → DELETE (mit Existenz-Check)
- ✓ PUT → UPDATE (optional, ähnlich zu POST)

Best Practices:

- ✓ **Routing-Pattern:** Nutzen Sie Router-Syntax wie `router.get('/products/:id', handler)`
- ✓ **Parameter-Extraktion:** Zugriff über `params.id` statt manueller Regex
- ✓ **Validierung:** Prüfen Sie Inputs, bevor Sie SQL ausführen
- ✓ **Error-Handling:** Nutzen Sie passende HTTP-Status Codes
- ✓ **RETURNING:** Bei INSERT/UPDATE/DELETE die geänderten Daten zurückgeben
- ✓ **Existenz-Checks:** Vor DELETE prüfen, ob Ressource existiert
- ⚠ **SQL-Injection:** In echten Systemen IMMER Prepared Statements nutzen!
- ⚠ **Transaktionen:** Bei Multi-Step-Operations (z.B. Bestellung + Items)

SQL-Injection Warnung:

```
// ✗ GEFÄHRlich (heute OK, weil nur lokal im Browser):  
const query = `SELECT * FROM users WHERE email = '${userInput}'`;  
  
// ✓ SICHER (echte Systeme):  
const query = 'SELECT * FROM users WHERE email = $1';  
const result = await db.query(query, [userInput]);
```

In echten Backends würden Sie niemals String-Interpolation nutzen! Heute geht es aber nur um das Konzept – die Daten bleiben im Browser.

Pro-Tipp: Lernen Sie ein echtes Backend-Framework (Express.js, FastAPI, Spring Boot) – die Konzepte von heute sind 1:1 übertragbar!

Referenzen & Weiterführendes

- **REST-API Design:** Roy Fielding's Dissertation (2000)
- **Fake Store API:** <https://fakestoreapi.com> (zum Üben)
- **MDN Web Docs:** Fetch API & HTTP-Methoden
- **OWASP:** SQL-Injection Prevention
- **HTTP-Status Codes:** RFC 7231