L18: GraphQL & SQL – Flexible Datenabfragen

Session 18 – Lecture

Dauer: 90 Minuten

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

Block: 4 – Theorie, Optimierung & Polyglot

Willkommen zur achtzehnten Session! In der letzten Vorlesung haben Sie gelernt, wie RESTful APIs HTTP-Requests in SQL-Queries übersetzen. Heute machen wir den nächsten Schritt: GraphQL! Eine moderne Alternative zu REST, die dem Client maximale Flexibilität gibt.

Stellen Sie sich vor: Statt mehrere REST-Endpoints anzufragen, schreiben Sie eine einzige Query, die exakt die Daten liefert, die Sie brauchen – nicht mehr, nicht weniger. Das ist GraphQL! Und das Beste: Ihre SQL-Skills sind direkt übertragbar.

Live-Demo: SpaceX GraphQL API

Bevor wir in die Theorie eintauchen, schauen wir uns ein echtes Beispiel an: Die öffentliche SpaceX GraphQL API!

```
// Öffentliche SpaceX GraphQL API abfragen (keine Auth nötig!)
const query = `
  query {
    company {
      name
      founder
      founded
      employees
      ceo
      cto
      summary
    }
    rockets(limit: 3) {
      name
      country
      first_flight
      cost_per_launch
      success_rate_pct
      engines {
        number
        type
      }
    }
  }
`;
```

```
  ;
26
27 const response = await fetch('https://spacex-production.up.railway.ap
28   method: 'POST',
29   headers: {
30     'Content-Type': 'application/json'
31   },
32   body: JSON.stringify({ query })
33 });
34
35 const result = await response.json();
36
37 console.log('🚀 Company Info:');
38 console.log(`   ${result.data.company.name} (gegründet ${result.data
     .company.founded})`);
39 console.log(`   CEO: ${result.data.company.ceo}, CTO: ${result.data.c
     .cto}`);
40 console.log(`   Mitarbeiter: ${result.data.company.employees}`);
41
42 console.log('\n🚀 Raketen:');
43 result.data.rockets.forEach(rocket => {
44   console.log(`\n   ${rocket.name} (${rocket.country})`);
45   console.log(`    • Erstflug: ${rocket.first_flight}`);
46   console.log(`    • Kosten: $${(rocket.cost_per_launch / 1000000).toF
       )}M`);
47   console.log(`    • Erfolgsrate: ${rocket.success_rate_pct}%`);
48   console.log(`    • Triebwerke: ${rocket.engines.number}× ${rocket.en
       .type}`);
49 });
```

```
🚀 Company Info:
   SpaceX (gegründet 2002)
   CEO: Elon Musk, CTO: Elon Musk
   Mitarbeiter: 9500

🚀 Raketen:

   Falcon 1 (Republic of the Marshall Islands)
   • Erstflug: 2006-03-24
   • Kosten: $6.7M
   • Erfolgsrate: 40%
   • Triebwerke: 1× merlin

   Falcon 9 (United States)
   • Erstflug: 2010-06-04
   • Kosten: $50.0M
   • Erfolgsrate: 98%
   • Triebwerke: 9× merlin

   Falcon Heavy (United States)
   • Erstflug: 2018-02-06
   • Kosten: $90.0M
   • Erfolgsrate: 100%
   • Triebwerke: 27× merlin
```

Sehen Sie? Eine Query, mehrere Ebenen tief (Company + Rockets + Engines) – bei REST wären das mindestens 4 separate Endpoints gewesen!

💡 **Weitere öffentliche GraphQL APIs:** Countries API, Rick and Morty API, GitHub API (mit Token)

Motivation: REST vs. GraphQL

Bevor wir in die Praxis einsteigen, schauen wir uns an, warum GraphQL entstanden ist und welche Probleme es löst.

Das Over-fetching Problem

Erinnern Sie sich an REST? Ein typisches Problem: Sie wollen nur den Namen und Preis eines Produkts, bekommen aber alle Felder.

REST-Request:

```
GET /products/5
```

REST-Response (zu viel!):

```
{
  "product_id": 5,
  "product_name": "Desk Chair",
  "price": 199.99,
  "description": "Ergonomic office chair...",
  "stock_quantity": 25,
  "supplier_id": 42,
  "created_at": "2024-01-10",
  "updated_at": "2024-02-01"
}
```

Sie brauchen nur Name und Preis, bekommen aber 8 Felder! Das verschwendet Bandbreite und Performance.

GraphQL-Alternative:

```
{
  product(id: 5) {
    name
    price
  }
}
```

GraphQL-Response (genau richtig!):

```
{
  "data": {
    "product": {
      "name": "Desk Chair",
      "price": 199.99
    }
  }
}
```

Sehen Sie den Unterschied? GraphQL gibt Ihnen exakt das, was Sie anfordern – nicht mehr, nicht weniger!

Das Under-fetching Problem

Zweites Problem bei REST: Verschachtelte Daten erfordern multiple Requests.

Szenario: Kunde mit seinen Bestellungen abrufen

REST-Approach (3 Requests!):

```
GET /customers/1              → Kunde
GET /customers/1/orders        → Bestellungen
GET /orders/101/items          → Bestellpositionen
```

Drei separate Requests! Das erhöht Latenz und Komplexität im Frontend-Code.

GraphQL-Alternative (1 Request!):

```
{
  customer(id: 1) {
    firstName
    lastName
    orders {
      orderDate
      totalAmount
      items {
        quantity
        product {
          name
          price
        }
      }
    }
  }
}
```

Eine Query, alle Daten! GraphQL löst verschachtelte Daten elegant auf – genau wie SQL-Joins!

Vergleichstabelle

Aspekt	REST	GraphQL
Endpoints	Multiple (`/products`, `/customers`)	Single (`/graphql`)
Data Fetching	Fixed structure	Flexible, client-defined
Over-fetching	Häufig (alle Felder)	Nein (nur gewünschte Felder)
Under-fetching	Multiple Requests nötig	Eine Query mit Nested Fields
Versionierung	URL-basiert (`/v1/`, `/v2/`)	Schema-Evolution (deprecation)
Dokumentation	Manuell (Swagger/OpenAPI)	Automatisch (introspection)
SQL-Übersetzung	Einfach (1:1 Mapping)	Komplex (Resolver + Query-Optimierung)
Caching	HTTP-basiert (einfach)	Komplex (benötigt Strategie)
Best Use Case	CRUD, einfache Ressourcen	Komplexe Daten-Graphen, Mobile Apps

GraphQL ist kein REST-Ersatz, sondern eine Alternative für komplexe Daten-Szenarien. Beide haben ihre Berechtigung!

Teil 1: GraphQL-Grundlagen

Jetzt lernen Sie die drei Kernkonzepte von GraphQL: Schema, Queries und Resolver.

Das Schema: Type System

GraphQL ist stark typisiert. Das Schema definiert, welche Daten verfügbar sind und welche Beziehungen existieren.

Beispiel-Schema (Online-Shop):

```
type Product {
  id: Int!
  name: String!
  price: Float!
  categories: [Category]
}

type Category {
  id: Int!
```

```
  id: Int!
  name: String!
  products: [Product]
}

type Customer {
  id: Int!
  firstName: String!
  lastName: String!
  email: String
  orders: [Order]
}

type Order {
  id: Int!
  orderDate: String
  totalAmount: Float
  items: [OrderItem]
}

type OrderItem {
  quantity: Int
  product: Product
  lineTotal: Float
}

type Query {
  products: [Product]
  product(id: Int!): Product
  customers: [Customer]
  customer(id: Int!): Customer
}
```

Wichtige Syntax: `!` bedeutet „required“ (NOT NULL in SQL), `[]` bedeutet Array/Liste. Das Schema ist wie ein ER-Diagramm – nur maschinenlesbar!

Schema = SQL-Schema Mapping:

GraphQL Type	SQL Equivalent
`type`	`TABLE`
`field`	`COLUMN`
`!`	`NOT NULL`
`[Type]`	One-to-Many Relation
`Type`	Many-to-One / Join

Sehen Sie die Parallele? GraphQL-Typen entsprechen Tabellen, Fields entsprechen Spalten, und Relationen werden durch verschachtelte Typen dargestellt!

Queries: Daten abfragen

Queries sind wie SELECT-Statements – aber mit verschachtelter Struktur.

Beispiel 1: Alle Produkte

```
{
  products {
    id
    name
    price
  }
}
```

SQL-Äquivalent:

```
SELECT product_id AS id,
       product_name AS name,
       price
FROM products;
```

Beispiel 2: Ein Produkt mit Kategorien (JOIN!)

```
{
  product(id: 1) {
    name
    price
    categories {
      name
    }
  }
}
```

SQL-Äquivalent:

```
-- Produkt
SELECT product_name, price FROM products WHERE product_id = 1;

-- Kategorien (separate Query!)
SELECT c.category_name
FROM categories c
INNER JOIN product_categories pc ON c.category_id = pc.category_id
WHERE pc.product_id = 1;
```

GraphQL-Queries sehen aus wie die Response-Struktur, die Sie zurückbekommen – das macht sie extrem intuitiv!

Resolver: SQL-Übersetzung

Resolver sind Funktionen, die GraphQL-Felder in SQL-Queries übersetzen. Das ist der Kern Ihrer Implementierung!

Resolver-Konzept:

```
const resolvers = {
  Query: {
    products: async () => {
      // SQL-Query ausführen
      const result = await db.query('SELECT * FROM products');
      return result.rows;
    },

    product: async (parent, args) => {
      // args.id kommt aus der GraphQL-Query
      const result = await db.query(
        'SELECT * FROM products WHERE product_id = $1',
        [args.id]
      );
      return result.rows[0];
    }
  },

  Product: {
    categories: async (parent) => {
      // parent enthält das Produkt-Objekt
      const result = await db.query(`
        SELECT c.*
        FROM categories c
        INNER JOIN product_categories pc ON c.category_id = pc.category_id
        WHERE pc.product_id = $1
      `, [parent.product_id]);
      return result.rows;
    }
  }
};
```

Verstehen Sie das Pattern? Jedes GraphQL-Feld bekommt einen Resolver, der eine SQL-Query ausführt. Verschachtelte Felder lösen weitere Queries aus!

Datenbank-Setup: Online-Shop

Wir nutzen das gleiche Schema wie in Lecture 17 – so können Sie REST und GraphQL direkt vergleichen!

```
1  -- Locations
```

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

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

-- 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)
);

-- 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
);

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

-- 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)
);

-- Order_Items
CREATE TABLE order_items (
```

```sql
  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)
);

-- Sample Data
INSERT INTO locations VALUES
  (1, 'Berlin', '10115', 'Germany'),
  (2, 'Hamburg', '20095', 'Germany'),
  (3, 'Munich', '80331', 'Germany');

INSERT INTO categories VALUES
  (1, 'Electronics', 'Electronic devices'),
  (2, 'Furniture', 'Office furniture'),
  (3, 'Accessories', 'Computer accessories');

INSERT INTO customers VALUES
  (1, 'Alice', 'Smith', 'alice@example.com', 'Main St', '42', 1),
  (2, 'Bob', 'Johnson', 'bob@example.com', 'Oak Ave', '15', 2),
  (3, 'Carol', 'Williams', 'carol@example.com', 'Elm St', '7', 3);

INSERT INTO products (product_name, price) VALUES
  ('Laptop', 999.99),
  ('Mouse', 29.99),
  ('Keyboard', 79.99),
  ('Monitor', 299.99),
  ('Desk Chair', 199.99),
  ('Webcam', 89.99),
  ('Headset', 149.99);

INSERT INTO product_categories VALUES
  (1, 1), (2, 1), (2, 3), (3, 1), (3, 3),
  (4, 1), (5, 2), (6, 1), (6, 3), (7, 1), (7, 3);

INSERT INTO orders VALUES
  (101, 1, '2024-01-15', 299.99, 'delivered'),
  (102, 2, '2024-01-22', 999.99, 'delivered'),
  (103, 1, '2024-02-01', 109.98, 'shipped'),
  (104, 3, '2024-02-03', 349.98, 'processing');

INSERT INTO order_items VALUES
  (1, 101, 4, 1, 299.99),
  (2, 102, 1, 1, 999.99),
  (3, 103, 2, 1, 29.99),
  (4, 103, 3, 1, 79.99),
  (5, 104, 5, 1, 199.99),
  (6, 104, 7, 1, 149.99);
```

```
-- 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'),
  (3, 'Munich', '80331', 'Germany')
```

ok

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

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),
  (3, 'Carol', 'Williams', 'carol@example.com', 'Elm St', '7', 3)
```

ok

```
INSERT INTO products (product_name, price) VALUES
  ('Laptop', 999.99),
  ('Mouse', 29.99),
  ('Keyboard', 79.99),
  ('Monitor', 299.99),
```

```
  ('Desk Chair', 199.99),
  ('Webcam', 89.99),
  ('Headset', 149.99)
```

ok

```
INSERT INTO product_categories VALUES
  (1, 1), (2, 1), (2, 3), (3, 1), (3, 3),
  (4, 1), (5, 2), (6, 1), (6, 3), (7, 1), (7, 3)
```

ok

```
INSERT INTO orders VALUES
  (101, 1, '2024-01-15', 299.99, 'delivered'),
  (102, 2, '2024-01-22', 999.99, 'delivered'),
  (103, 1, '2024-02-01', 109.98, 'shipped'),
  (104, 3, '2024-02-03', 349.98, 'processing')
```

ok

```
INSERT INTO order_items VALUES
  (1, 101, 4, 1, 299.99),
  (2, 102, 1, 1, 999.99),
  (3, 103, 2, 1, 29.99),
  (4, 103, 3, 1, 79.99),
  (5, 104, 5, 1, 199.99),
  (6, 104, 7, 1, 149.99)
```

ok

Perfekt! Unsere Datenbank ist bereit. Jetzt bauen wir die GraphQL-Schicht!

Teil 2: GraphQL-Setup mit PGlite

Jetzt implementieren wir eine vollständige GraphQL-Schicht – direkt im Browser mit PGlite!

Schema definieren

Zuerst erstellen wir das GraphQL-Schema. Es beschreibt alle verfügbaren Typen und Queries.

```graphql
type Product {
  id: Int!
  name: String!
  price: Float!
  categories: [Category]
}

type Category {
  id: Int!
```

```
  id: Int!
  name: String!
  description: String
}

type Customer {
  id: Int!
  firstName: String!
  lastName: String!
  email: String
  location: Location
  orders: [Order]
}

type Location {
  id: Int!
  city: String!
  postalCode: String!
  country: String!
}

type Order {
  id: Int!
  orderDate: String
  totalAmount: Float
  status: String
  items: [OrderItem]
}

type OrderItem {
  quantity: Int
  lineTotal: Float
  product: Product
}

type Query {
  products: [Product]
  product(id: Int!): Product
  categories: [Category]
  customers: [Customer]
  customer(id: Int!): Customer
}
```

```
✅ Schema created
undefined
```

Das Schema ist wie ein Vertrag: Es definiert, welche Daten der Client abfragen kann und welche Typen existieren.

Resolver mit PGlite implementieren

Jetzt kommt der spannende Teil: Wir verbinden GraphQL mit SQL!

```
// Root Resolver (Top-Level Queries)
window.rootResolver = {
  // Alle Produkte
  products: async () => {
    const result = await db.query(`
      SELECT product_id AS id,
             product_name AS name,
             price
      FROM products
      ORDER BY product_name
    `);
    return result.rows;
  },

  // Ein Produkt nach ID
  product: async ({ id }) => {
    const result = await db.query(`
      SELECT product_id AS id,
             product_name AS name,
             price
      FROM products
      WHERE product_id = $1
    `, [id]);
    return result.rows[0];
  },

  // Alle Kategorien
  categories: async () => {
    const result = await db.query(`
      SELECT category_id AS id,
             category_name AS name,
             description
      FROM categories
      ORDER BY category_name
    `);
    return result.rows;
  },

  // Alle Kunden
  customers: async () => {
    const result = await db.query(`
      SELECT customer_id AS id,
             first_name AS "firstName",
             last_name AS "lastName",
             email,
             location_id AS "locationId"
```

```
      FROM customers
      ORDER BY last_name, first_name
    `);
    return result.rows;
  },

  // Ein Kunde nach ID
  customer: async ({ id }) => {
    const result = await db.query(`
      SELECT customer_id AS id,
             first_name AS "firstName",
             last_name AS "lastName",
             email,
             location_id AS "locationId"
      FROM customers
      WHERE customer_id = $1
    `, [id]);
    return result.rows[0];
  }
};

console.log('✅ Resolver created');
```

```
✅ Resolver created
```

Sehen Sie das Pattern? Jeder Resolver führt eine SQL-Query aus und gibt das Ergebnis zurück. GraphQL kümmert sich um den Rest!

Erste GraphQL-Query ausführen

Jetzt testen wir unsere Implementierung mit einer einfachen Query!

```
{
  products {
    id
    name
    price
  }
}
```

```
{
  "data": {
    "products": [
      {
        "id": 5,
        "name": "Desk Chair",
        "price": 199.99
      },
      {
        "id": 7,
        "name": "Headset",
        "price": 149.99
      },
      {
        "id": 3,
        "name": "Keyboard",
        "price": 79.99
      },
      {
        "id": 1,
        "name": "Laptop",
        "price": 999.99
      },
      {
        "id": 4,
        "name": "Monitor",
        "price": 299.99
      },
      {
        "id": 2,
        "name": "Mouse",
        "price": 29.99
      },
      {
        "id": 6,
        "name": "Webcam",
        "price": 89.99
      }
    ]
  }
}
```

Fantastisch! GraphQL hat die Query ausgeführt, unseren Resolver aufgerufen, der eine SQL-Query ausgeführt hat – und das Ergebnis zurückgegeben!

Teil 3: Nested Resolver (Joins!)

Jetzt wird es spannend: Verschachtelte Daten! Das ist die Stärke von GraphQL.

Lösung: Field Resolver mit parent-Objekt

Die elegante Lösung: Field-Resolver! GraphQL übergibt automatisch das Parent-Objekt, sodass wir `parent.id` für SQL-Queries nutzen können.

```
// Field-Resolver: GraphQL ruft sie automatisch auf!
window.nestedResolver = {
  // Query-Resolver (Top-Level) - nur eine SQL-Query!
  ...rootResolver,

  // Field-Resolver: Werden automatisch von GraphQL aufgerufen!
  Product: {
    categories: async (parent) => {
      // parent = { id: 1, name: "Laptop", price: 999.99 }
      console.log(`🔍 Loading categories for product ${parent.id} (${
        .name})...`);
      const result = await db.query(`
        SELECT c.category_id AS id,
               c.category_name AS name,
               c.description
        FROM categories c
        INNER JOIN product_categories pc ON c.category_id = pc.catego
        WHERE pc.product_id = $1
      `, [parent.id]);
      console.log(`  ✅ Found ${result.rows.length} categories for ${
        .name}`);
      return result.rows;
    }
  },

  Customer: {
    location: async (parent) => {
      if (!parent.locationId) return null;

      const result = await db.query(`
        SELECT location_id AS id,
               city,
               postal_code AS "postalCode",
               country
        FROM locations
        WHERE location_id = $1
      `, [parent.locationId]);
      return result.rows[0];
```

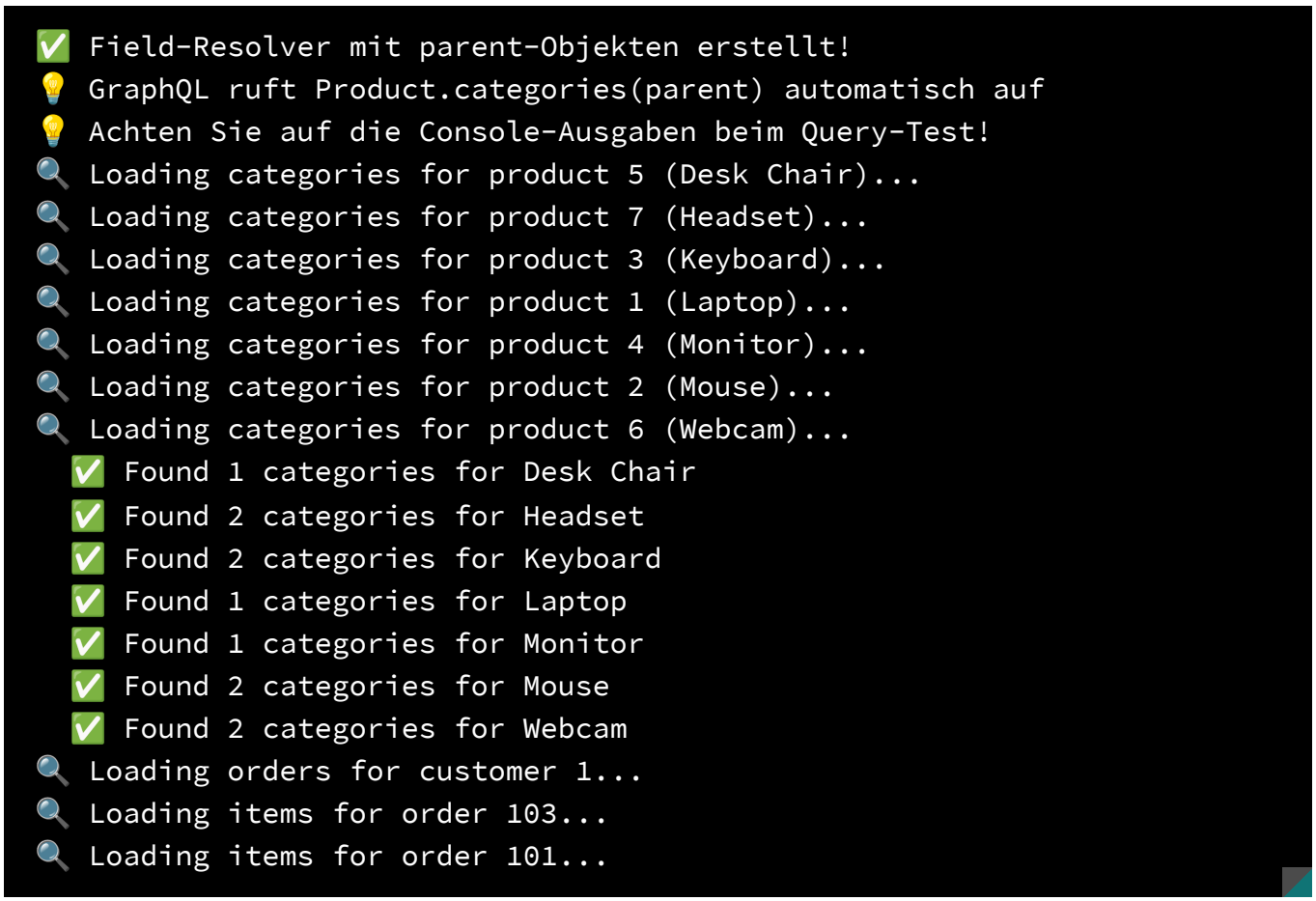
```
    },

    orders: async (parent) => {
      console.log(`🔍 Loading orders for customer ${parent.id}...`);
      const result = await db.query(`
        SELECT order_id AS id,
               order_date AS "orderDate",
               total_amount AS "totalAmount",
               status
        FROM orders
        WHERE customer_id = $1
        ORDER BY order_date DESC
      `, [parent.id]);
      return result.rows;
    }
  },

  Order: {
    items: async (parent) => {
      console.log(`🔍 Loading items for order ${parent.id}...`);
      const result = await db.query(`
        SELECT oi.quantity,
               oi.line_total AS "lineTotal",
               oi.product_id AS "productId"
        FROM order_items oi
        WHERE oi.order_id = $1
      `, [parent.id]);
      return result.rows;
    }
  },

  OrderItem: {
    product: async (parent) => {
      const result = await db.query(`
        SELECT product_id AS id,
               product_name AS name,
               price
        FROM products
        WHERE product_id = $1
      `, [parent.productId]);
      return result.rows[0];
    }
  }
};

console.log('✅ Field-Resolver mit parent-Objekten erstellt!');
console.log('💡 GraphQL ruft Product.categories(parent) automatisch a
console.log('💡 Achten Sie auf die Console-Ausgaben beim Query-Test!'
```



Das ist die elegante Lösung! Jeder Resolver hat nur eine SQL-Query. GraphQL ruft die Field-Resolver automatisch auf und übergibt das Parent-Objekt. So funktioniert es in Production!

Query: Produkte mit Kategorien

Testen wir es: Produkte mit ihren Kategorien abrufen. Achten Sie auf die Console – Sie sehen die Field-Resolver in Aktion!

```
{
  products {
    name
    price
    categories {
      name
    }
  }
}
```

```
{
  "data": {
    "products": [
      {
        "name": "Desk Chair",
        "price": 199.99,
        "categories": [
          {
            "name": "Furniture"
          }
        ]
      },
      {
        "name": "Headset",
        "price": 149.99,
        "categories": [
          {
            "name": "Electronics"
          },
          {
            "name": "Accessories"
          }
        ]
      },
      {
        "name": "Keyboard",
        "price": 79.99,
        "categories": [
          {
            "name": "Electronics"
          },
          {
            "name": "Accessories"
          }
        ]
      },
      {
        "name": "Laptop",
        "price": 999.99,
        "categories": [
          {
            "name": "Electronics"
          }
```

```
        ]
      },
      {
        "name": "Monitor",
        "price": 299.99,
        "categories": [
          {
            "name": "Electronics"
          }
        ]
      },
      {
        "name": "Mouse",
        "price": 29.99,
        "categories": [
          {
            "name": "Electronics"
          },
          {
            "name": "Accessories"
          }
        ]
      },
      {
        "name": "Webcam",
        "price": 89.99,
        "categories": [
          {
            "name": "Electronics"
          },
          {
            "name": "Accessories"
          }
        ]
      }
    ]
  }
}
```


Sehen Sie das N+1-Problem? Für 7 Produkte haben wir 8 SQL-Queries ausgeführt! Jedes Produkt löst eine separate Kategorien-Query aus Perfekt! GraphQL hat automatisch die Resolver-Kette aufgerufen: erst `products`, dann für jedes Produkt `categories`.

Query: Kunde mit Bestellungen (Deep Nesting!)

Jetzt testen wir eine tief verschachtelte Query – das ist die wahre Stärke von GraphQL!

```
{
  customer(id: 1) {
    firstName
    lastName
    email
    location {
      city
      country
    }
    orders {
      orderDate
      totalAmount
      status
      items {
        quantity
        lineTotal
        product {
          name
          price
        }
      }
    }
  }
}
```

```
{
  "data": {
    "customer": {
      "firstName": "Alice",
      "lastName": "Smith",
      "email": "alice@example.com",
      "location": {
        "city": "Berlin",
        "country": "Germany"
      },
      "orders": [
        {
          "orderDate": "1706745600000",
          "totalAmount": 109.98,
          "status": "shipped",
          "items": [
            {
              "quantity": 1,
              "lineTotal": 29.99,
              "product": {
                "name": "Mouse",
                "price": 29.99
              }
            },
            {
              "quantity": 1,
              "lineTotal": 79.99,
              "product": {
                "name": "Keyboard",
                "price": 79.99
              }
            }
          ]
        },
        {
          "orderDate": "1705276800000",
          "totalAmount": 299.99,
          "status": "delivered",
          "items": [
            {
              "quantity": 1,
              "lineTotal": 299.99,
              "product": {
```

```
                "name": "Monitor",
                "price": 299.99
              }
            }
          ]
        }
      ]
    }
  }
}
```

Beeindruckend! Eine Query, fünf Ebenen tief – aber achten Sie auf die Anzahl der SQL-Queries. Jede Verschachtelungsebene löst weitere Queries aus. Das ist das N+1-Problem in Aktion –{{1}}– Beeindruckend! Eine Query, fünf Ebenen tief – und GraphQL hat automatisch alle SQL-Queries koordiniert!

Teil 4: Mutations & Erweiterungen

Bisher haben Sie nur Queries (READ) implementiert. Jetzt lernen Sie Mutations – das GraphQL-Äquivalent zu INSERT, UPDATE und DELETE!

Schema mit Mutations erweitern

Mutations sind Operationen, die Daten verändern. Sie werden im Schema separat definiert.

```
type Mutation {
  createProduct(name: String!, price: Float!): Product
  updateProduct(id: Int!, name: String, price: Float): Product
  deleteProduct(id: Int!): Boolean
}
```

```
✅ Schema mit Mutations erstellt
undefined
```

Mutations sind wie Query-Resolver – nur dass sie Daten verändern statt nur zu lesen!

Mutation Resolver implementieren

Jetzt implementieren wir die Resolver für CREATE, UPDATE und DELETE.

```
window.resolverWithMutations = {
  // Query Resolver (wie vorher)
  ...nestedResolver,

  // Mutation Resolver (NEU!)
  createProduct: async ({ name, price }) => {
    console.log(`📝 Creating product: ${name} (${price}€)`);
```

```
    const result = await db.query(`
      INSERT INTO products (product_name, price)
      VALUES ($1, $2)
      RETURNING product_id AS id, product_name AS name, price
    `, [name, price]);

    console.log(`✅ Product created with ID ${result.rows[0].id}`);
    return result.rows[0];
  },

  updateProduct: async ({ id, name, price }) => {
    console.log(`✏️ Updating product ID ${id}...`);

    // Dynamisches UPDATE: nur Felder aktualisieren, die übergeben wu
    const updates = [];
    const params = [];
    let paramCount = 1;

    if (name !== undefined) {
      updates.push(`product_name = $${paramCount++}`);
      params.push(name);
    }
    if (price !== undefined) {
      updates.push(`price = $${paramCount++}`);
      params.push(price);
    }

    if (updates.length === 0) {
      throw new Error('Keine Updates angegeben');
    }

    params.push(id);

    const result = await db.query(`
      UPDATE products
      SET ${updates.join(', ')}
      WHERE product_id = $${paramCount}
      RETURNING product_id AS id, product_name AS name, price
    `, params);

    if (result.rows.length === 0) {
      throw new Error(`Product ID ${id} nicht gefunden`);
    }

    console.log(`✅ Product updated: ${result.rows[0].name}`);
    return result.rows[0];
  },

  deleteProduct: async ({ id }) => {
    console.log(`🗑️ Deleting product ID ${id}...`);
```

```
59
60      // Erst prüfen, ob Produkt existiert
61      const checkResult = await db.query(`
62        SELECT product_name FROM products WHERE product_id = $1
63      `, [id]);
64
65      if (checkResult.rows.length === 0) {
66        throw new Error(`Product ID ${id} nicht gefunden`);
67      }
68
69      const productName = checkResult.rows[0].product_name;
70
71      // Produkt löschen
72      await db.query(`
73        DELETE FROM products WHERE product_id = $1
74      `, [id]);
75
76      console.log(`✅ Product "${productName}" deleted`);
77      return true;
78    },
79
80    // Field-Resolver für verschachtelte Daten
81    Product: {
82      categories: async (parent) => {
83        const result = await db.query(`
84          SELECT c.category_id AS id,
85                 c.category_name AS name,
86                 c.description
87          FROM categories c
88          INNER JOIN product_categories pc ON c.category_id = pc.catego
89          WHERE pc.product_id = $1
90        `, [parent.id]);
91        return result.rows;
92      }
93    }
94  };
95
96  console.log('✅ Resolver mit Mutations erstellt');
97  console.log('💡 Jetzt können Sie Produkte erstellen, aktualisieren un
      löschen!');
```

```
✅ Resolver mit Mutations erstellt
💡 Jetzt können Sie Produkte erstellen, aktualisieren und löschen!
📝 Creating product: External SSD 1TB (129.99€)
✅ Product created with ID 8
✏️ Updating product ID 1...
✅ Product updated: Gaming Laptop
🗑️ Deleting product ID 8...
✅ Product "External SSD 1TB" deleted
```

Sehen Sie das Pattern? Mutations sind normale Resolver-Funktionen – aber sie führen INSERT, UPDATE oder DELETE aus statt SELECT!

Mutation: Produkt erstellen (CREATE)

Testen wir die erste Mutation: Ein neues Produkt anlegen!

```
mutation {
  createProduct(name: "External SSD 1TB", price: 129.99) {
    id
    name
    price
  }
}
```

```
📊 Result:
{
  "data": {
    "createProduct": {
      "id": 8,
      "name": "External SSD 1TB",
      "price": 129.99
    }
  }
}
```

Perfekt! GraphQL hat das neue Produkt mit INSERT erstellt und die ID zurückgegeben!

Mutation: Produkt aktualisieren (UPDATE)

Jetzt aktualisieren wir ein bestehendes Produkt. Sie können einzelne Felder oder mehrere gleichzeitig ändern!

```
mutation {
  updateProduct(id: 1, name: "Gaming Laptop", price: 1299.99) {
```

```
    id
    name
    price
  }
}
```

```
📊 Result:
{
  "data": {
    "updateProduct": {
      "id": 1,
      "name": "Gaming Laptop",
      "price": 1299.99
    }
  }
}
```

Super! Das UPDATE hat funktioniert. Die dynamische Query erlaubt es, nur die Felder zu ändern, die Sie übergeben!

Mutation: Produkt löschen (DELETE)

Zum Schluss löschen wir ein Produkt. Vorsicht: Diese Operation ist nicht umkehrbar!

```
mutation {
  deleteProduct(id: 8)
}
```

```
{
  "data": {
    "deleteProduct": true
  }
}
```

Gelöscht! Der DELETE-Resolver prüft erst, ob das Produkt existiert, und löscht es dann aus der Datenbank.

GraphQL vs. SQL: Mutations

Vergleich der Operationen:

GraphQL Mutation	SQL Equivalent
`createProduct(...)`	`INSERT INTO products ...`
`updateProduct(...)`	`UPDATE products SET ... WHERE`
`deleteProduct(...)`	`DELETE FROM products WHERE`

Der große Vorteil: GraphQL gibt Ihnen strukturierte Responses zurück – Sie können direkt abfragen, welche Felder Sie nach der Mutation zurückhaben möchten!

GraphQL ↔ SQL Mapping (Kern-Takeaways)

```
GraphQL Type       →  SQL Table
GraphQL Field      →  SQL Column
GraphQL Relation   →  SQL Foreign Key + JOIN
GraphQL Query      →  SELECT (mit dynamischen WHERE/JOIN)
GraphQL Mutation   →  INSERT / UPDATE / DELETE
Resolver Function  →  SQL Query Executor
```

Wann GraphQL, wann REST?

Nutzen Sie REST, wenn:

- Einfache CRUD-Operationen
- HTTP-Caching wichtig ist
- Backend-Kontrolle über Datenstruktur gewünscht
- Team mit REST-Expertise

Nutzen Sie GraphQL, wenn:

- Komplexe, verschachtelte Daten
- Mobile Apps (Bandbreiten-Optimierung)
- Verschiedene Clients mit unterschiedlichen Anforderungen
- Stark typgef Schnittstelle gewünscht
- Rapid Prototyping (selbst-dokumentierendes Schema)