

8.3 Das initiale Schema aufsetzen

Nun da der GraphQL-Server erfolgreich initialisiert ist, kann das Schema wie in Kapitel 3.3 erweitert werden. Hierfür sind zuerst die Typdefinitionen der Objekte vom Use Case abzuleiten. Anschließend muss der Resolver entsprechend angepasst werden, um die neuen Querys erfolgreich aufzulösen.

Aus dem Use Case lässt sich ableiten, dass der GraphShop aus drei Elementen besteht: Kunden, Adressen und Bestellungen, dabei ist zu beachten, dass ein Kunde mehrere Adressen und Bestellungen hat. Zudem hält eine Bestellung immer exakt eine Adresse.

8.3.1 Objekte im Schema auflösen

In der bisherigen Schemadefinition besteht der Kunde bisher nur aus einem String für seinen Namen. Es gilt, einen neuen Customer-Typ zu definieren, um diesen später mit Feldern zu erweitern.

Listing 8-5
Der Costumer-Typ in
`schema.graphqls`

```
type Query {
  customer: Customer,
}
```

```
type Customer {
  fullname: String!
}
```

Um diesen neuen Typ auch im Resolver innerhalb des *java*-Moduls nutzen zu können, muss die Definition dort gespiegelt werden. Hierfür kann ein einfaches Plain Old Java Object (POJO) zum Einsatz kommen. In diesem Beispiel ist dafür innerhalb eines neuen *model*-Packages die Customer-Klasse zu erzeugen.

Listing 8-6
Die
Customer-POJO-Klasse

```
package com.example.graphshop.model;

public class Customer {
  private String firstname;
  private String lastname;

  public Customer (String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  public String getFirstname() {
    return firstname;
  }
}
```

```

public String getLastName() {
    return lastname;
}
}

```

Um auf Kunden zugreifen zu können und eventuelle Geschäftslogik umzusetzen, muss noch ein Service für diese neue Klasse geschrieben werden. Dieser würde normalerweise über ein *Repository* den Zugriff auf die Daten erhalten. In diesem Beispiel machen wir es uns einfach und persistieren die Beispieldaten direkt innerhalb des Services.

Um der bisherigen Struktur zu folgen, wird der neue *Customer-Service* in einem entsprechenden *service*-Package erstellt.

```

package com.example.graphshop.service;

import com.example.graphshop.model.Customer;
import org.springframework.stereotype.Service;

@Service
public class CustomerService {

    private Customer customerData =
        new Customer("Luke", "Skywalker");

    public Customer getCustomer(){
        return customerData;
    }
}

```

Nun ist der *CustomerService* noch im Resolver zu registrieren. Anschließend kann auch der Aufruf des Kunden im Schema über die Rückgabe des *Customer*-POJOs aus dem Service aufgelöst werden.

```

@Service
public class QueryResolver
    implements GraphQLQueryResolver {

    private CustomerService customerService;

    public QueryResolver(
        final CustomerService customerService) {
        this.customerService = customerService;
    }
}

```

Listing 8-7

Der *Customer*-Typ in *schema.graphqls*

Listing 8-8

Das *Customer*-Feld gegen den *Customer-Service* auflösen

```

public Customer customer() {
    return customerService.getCustomer();
}
}

```

8.3.2 Feld-Resolver

Eine Abfrage des `fullname`-Feldes aus dem `Customer`-Typ ist damit jedoch noch nicht möglich. Da das `Customer`-POJO kein entsprechendes Feld für den vollständigen Namen, sondern nur für Vor- und Nachnamen getrennt besitzt, können die Standard-Feld-Resolver das entsprechende Feld nicht auflösen.

Bei dieser Diskrepanz zwischen bereits existierendem Datensatz und der idealen GraphQL-Schemadefinition helfen sogenannte Feld-Resolver. Diese lösen einzelne Felder eines definierten Typs gezielt auf. Sie erlauben somit sogenannte *Computed Fields*, also Felder mit Geschäftslogik.

Für den Kunden muss der *CustomerResolver* innerhalb des *resolver*-Packages erzeugt werden. Dieser implementiert das *GraphQLResolver*-Interface. Das benötigt zusätzlich noch einen *Generic Type* des Objekts, für das es Felder auflösen soll, in diesem Fall *Customer*.

Listing 8–9

Der
Customer-Feld-Resolver

```

@Service
public class CustomerResolver
    implements GraphQLResolver<Customer> {

    public String fullname (final Customer customer) {
        return customer.getFirstname() + " " +
            customer.getLastname();
    }
}

```

Die Methode zum Auflösen des Feldes bekommt das aufgerufene Objekt als Parameter übergeben. Mit diesem POJO kann anschließend die erforderte Logik für das Feld im GraphQL-Schema implementiert werden.

Mit einem Neustart der Applikation kann nun wie in Abbildung 8–3 über GraphQL das `fullname`-Feld des `customer`-Typs aufgerufen werden. Trotz `firstname`- und `lastname`-Feldern in der Datenschicht wird es nun richtig aufgelöst.

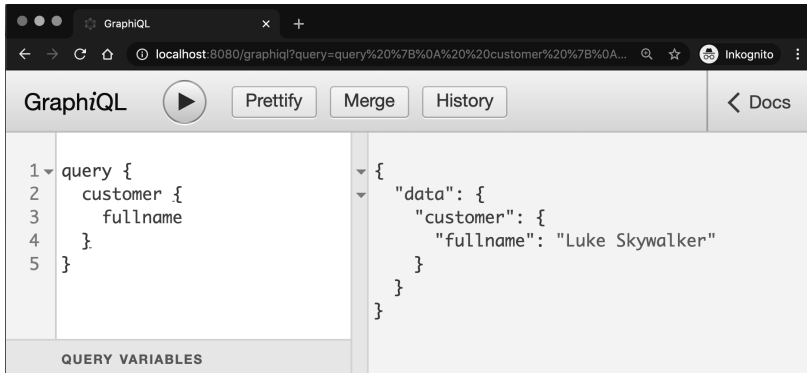


Abb. 8-3
Customer-Fullname-
Abfrage mit GraphQL

8.3.3 Das Node-Muster

Da Kunden – und auch alle weiteren Objekte – laut Use Case eine ID benötigen, ist dies eine gute Stelle, um das Node-Muster (siehe Kapitel 4.2.1) einzuführen. Hierfür muss zuerst die Definition des Node-Interface im Schema erfolgen. Anschließend kann die Typdefinition des Kunden als Implementierung eines Nodes definiert werden.

```
type Query {
  customer: Customer
}
interface Node {
  id: ID!
}
type Customer implements Node {
  id: ID!
  fullname: String!
}
```

Listing 8-10
Einführen des
Node-Musters in
schema.graphqls

Als Nächstes muss diese Definition auch im *java*-Modul abgebildet werden. Hierfür wird im *model*-Package das Interface *Node* erzeugt. Das kann zwar – anders als die Typdefinition im GraphQL-Schema – keine Felder halten, jedoch wird die *getId*-Methode definiert, die später Implementierungen des Interface zu einem entsprechenden ID-Feld erzwingt.

```
package com.example.graphshop.model;

public interface Node {
  String getId();
}
```

Listing 8-11
Implementieren des
Java-Node-Interface

Die eindeutige ID, die jede Implementierung von Node benötigt, könnte einfach über die Java-eigene *UUID*-Hilfsmethode implementiert werden. Um die manuellen Tests jedoch einfacher zu halten, wird in diesem Beispiel eine simple Hilfsklasse erzeugt, die diese Arbeit erledigt. Dafür wird in einem neuen *utils*-Package die folgende *UUID*-Klasse erzeugt.

Listing 8-12
Implementieren
der eigenen
UUID-Hilfsklasse

```
package com.example.graphshop.utils;

public class UUID {
    private static int id = 0;

    public static String getId() {
        return String.valueOf(id++);
    }
}
```

Zum Abschluss fehlt nun nur noch die Erweiterung des *Customer*-POJO als Implementierung dieses Interfaces. Dafür ist das *id*-Feld einzuführen, das im Konstruktor der Klasse mit einer eindeutigen ID über die gerade erzeugte *UUID*-Klasse gefüllt wird. Anschließend muss der im Interface definierte *Getter* des *id*-Feldes implementiert werden.

Listing 8-13
Definition der
Customer-Klasse als
Implementierung des
Node-Interface

```
package com.example.graphshop.model;

import com.example.graphshop.utils.UUID;

public class Customer implements Node {
    private String id;
    private String firstname;
    private String lastname;

    public Customer (String firstname, String lastname) {
        this.id = UUID.getId();
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public String getId() {
        return id;
    }

    ...
}
```

Ein Neustart der Applikation sowie eine Abfrage des neuen ID-Felds durch GraphQL wie in Abbildung 8-4 lösen nun ebenfalls die erzeugte, eindeutige ID des Kunden auf.

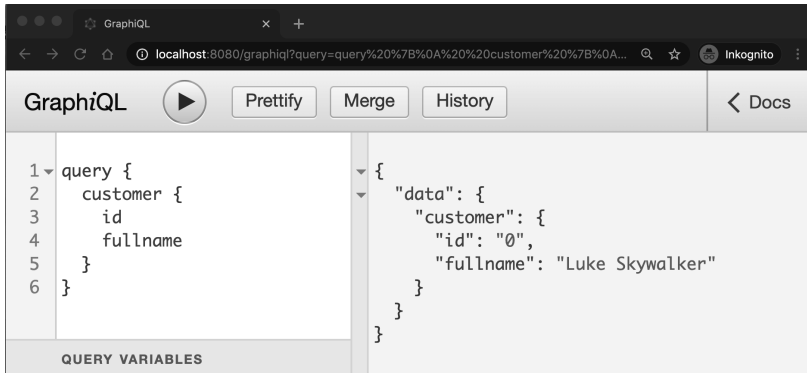


Abb. 8-4
Customer-ID-Abfrage
mit GraphQL

8.3.4 Ergebnisse filtern durch Parameter

Der Beispieldatensatz hat noch ein Problem: Üblicherweise gibt es in einem Onlineshop nicht nur einen Kunden, sondern viele. Um Informationen über einen einzelnen Kunden zu erhalten, muss diese Liste gefiltert werden.

Um das abzubilden, bedarf es einer Anpassung der Beispieldaten im *CustomerService*. Statt eines einzelnen Kunden wird hier nun eine Liste mehrerer Kunden persistiert. Der Zugriff in dieser Liste erfolgt lediglich auf einzelne Kunden, identifiziert durch ihre eindeutige ID aus einem mitgelieferten Parameter.

```
@Service
public class CustomerService {

    private List<Customer> customerData = new ArrayList<>(
        Arrays.asList(new Customer("Luke", "Skywalker"),
            new Customer("Leia", "Organa")));

    public Customer getCustomer(String id) {
        return customerData.stream()
            .filter((c) -> c.getId().equals(id))
            .findFirst()
            .orElse(null);
    }
}
```

Listing 8-14
Erweiterung des
Beispieldatensatzes im
CustomerService

Um diese ID aus der Query heraus an den Service durchreichen zu können, muss sie natürlich auch als Parameter in das Schema eingeführt werden. Hierzu ist der *customer*-Feldaufruf in der Root-Query zu bearbeiten.

Listing 8-15
ID-Parameter im
Schema einführen

```
type Query {
  customer(id: ID!): Customer,
}
```

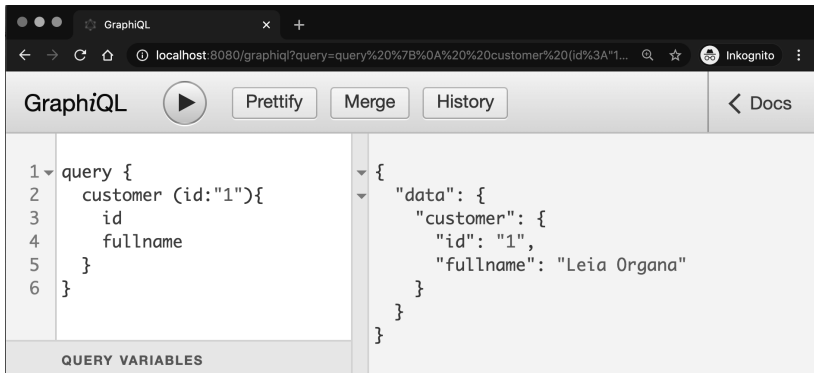
Da es nur einen einzelnen Rückgabewert aus der Liste geben kann, muss der ID-Parameter mit dem Non-Null-Modifikator versehen werden. Um diese Query entsprechend auflösen zu können, erfolgt anschließend die Erweiterung des *QueryResolvers* um den Parameter sowie das korrekte Weiterreichen an den Service.

Listing 8-16
QueryResolver um
ID-Parameter erweitern

```
public Customer customer(String id) {
  return customerService.getCustomer(id);
}
```

Nun kann der Test mit dem neuen Parameter erfolgen. Wird die Query wie bisher ohne Argumente ausgeführt, sollte sie jetzt einen Typ-Error ausgeben. Mit einem beigefügten Parameter wie in Abbildung 8-5 kann jedoch ein spezifischer Kunde angefragt werden.

Abb. 8-5
Customer-ID-
Parameter-Abfrage
mit GraphQL



8.3.5 Objekt-Relationen

Neben dem Namen und der ID hält ein Kunde laut dem Use Case noch eine Liste an Adressen. Um das abzubilden, muss eine neue Typdefinition in das Schema aufgenommen werden, vorerst in einer minimalen Version mit nur einem repräsentativen Feld.

Listing 8-17
Schemaerweiterung
um Address-Typ

```
type Customer implements Node {
  id: ID!
  fullname: String!
  addresses: [Address!]!
}
```