Damit wird ein Fokus nicht nur auf einzelne Elemente, sondern auf deren Relation zueinander gelegt. Diese ist nicht mehr lose gekoppelt wie bei REST: Mit einem einzelnen Einstiegspunkt erreicht man einige Elemente unter Umständen nur, indem man den Weg der Relationen des Elements zum Einstiegspunkt zurückverfolgt.

Die komplette Geschäftsdomäne, in der man sich befindet, wird am einfachsten über verschiedene Knotenpunkte im Objektbaum, sogenannten Nodes, und deren Relationen zueinander erklärt. Die Definition der Nodes und ihrer Verbindungen dient dann bereits als Schema für die Datenstruktur, auf der GraphQL operiert.

Dieses Schema kann dann für jegliche Technologie der unterliegenden Backend-Systeme, also die Datenbank und der datenverarbeitenden Server darüber, in die jeweilige Implementierung der GraphQL-Services übersetzt werden. Das Schema selbst definiert ja nur das Interface des Systems nach außen. Dies bedeutet also: Erst muss man sich über die Datenstruktur – den Nodes und vor allem der Relationen dieser Nodes – innerhalb der eigenen Geschäftsdomäne klar sein.

Dieser Fokus auf die Beziehungen der Elemente statt nur auf die der einzelnen Nodes bedeutet eine klare Bedingung oder auch Einschränkung für die Datenstruktur: Es müssen Relationen vorliegen. Umso verzweigter die Verbindungen in der Datenstruktur sind – umso tiefer also der Graph ist, mit dem man arbeitet –, desto mehr profitiert man von den Möglichkeiten einer GraphQL API.

Grundlegend bedeutet das jedoch auch: Umso einfacher die Datenstruktur ist und damit auch umso flacher der Graph an Daten ist, desto weniger hat man von der eigentlichen Stärke eines solchen API. Viele einzeln stehende, gekapselte Ressourcen sind eventuell nicht mehr den Overhead einer Schemadefinition mit GraphQL wert und wären daher mit einem simplen REST API besser bedient.

Möchte man aber eine Schnittstelle für komplexe Datenstrukturen mit vielen Abhängigkeiten und Relationen erzeugen, dann bietet GraphQL eine Liste an Werkzeugen, die immense Vorteile gegenüber anderen Herangehensweisen bieten. Wie diese aussehen und damit auch, wie man mit GraphQL im Detail arbeitet, wird in diesem Kapitel behandelt.

3.1 Das Graphen-Modell erzeugen

Da die Datenstruktur für GraphQL ein entscheidender Baustein ist, sollte man sich als Allererstes Gedanken darüber machen. Wichtig ist hierbei ein Fokus aufs Wesentliche: Ein Schema besteht aus Nodes und ihren Relationen. Geschäftslogik sollte idealerweise auf dem Server aus-

geführt werden. GraphQL ist lediglich der Zugriff auf diesen. Validierung, Fehlerbehandlung und Autorisierung sollten also auf dem Server an einem zentralen Ort erfolgen, sonst besteht die Gefahr, nicht durchgehend einheitliche Geschäftslogik im System zu pflegen. Das ist vor allem wichtig bei Systemen, die mehrere Einstiegspunkte in die Datenstruktur oder gleich verschiedene Schemadefinitionen besitzen – beispielsweise für eine spezielle Kampagnenansicht für Kunden und eine andere für Marketingmanager.

Das Schema besteht also nur aus Ressourcen und Relationen. Innerhalb dieser Datenstruktur ist man jedoch sehr flexibel. Wenn man beispielsweise ein Schema für ein bestehendes Altsystem erstellen möchte, muss man keinesfalls die bereits existierende Datenstruktur abbilden, sondern kann die perfekte Repräsentation für die Nutzung der Daten durch die konsumierenden Entwickler aufbauen. GraphQL agiert lediglich als Zwischenschicht, sozusagen eine Übersetzung der Schnittstelle zum bestehenden System. Das bedeutet also, dass man die Datenstruktur, die letztendlich am Interface liegt, anpassen kann, ohne dabei das bestehende System ändern zu müssen.

Das Schema sollte immer eine Beschreibung der Daten sein, die es repräsentiert, und nicht, wie diese Daten im System strukturiert sind. Das hilft dabei, den Fokus auf den konsumierenden Entwickler und den letztendlichen Endkunden zu legen und nicht auf das dahinterliegende System.

Um diesen Fokus über die komplette Schemadefinition zu validieren, ist es vor allem wichtig, immer wieder Rücksprache mit den potenziellen Nutzern des Interfaces zu halten, wenn das möglich ist. Ein komplettes Schema für eine ganze Geschäftsdomäne in einem Durchgang zu erstellen, führt oft zu unnötig komplizierten Lösungen oder übersieht einige Anforderungen.

Viel zielführender ist es, sich an einzelnen Use Cases zu orientieren (siehe Kapitel 2.3.1), das Schema für diesen einen Case zu definieren, Feedback einzuholen und das Schema dann zu optimieren. Ist die Validierung des Schemas durch potenzielle Nutzer erfolgt, kann man dieses durch weitere Szenarien erweitern. Auch hier sollte wieder Feedback eingeholt und das Schema entsprechend optimiert werden. Diese graduelle Erweiterung des Schemas sichert, dass alle Prozesse in der Geschäftsdomäne klar abgebildet sind, und führt dazu, dass es möglichst intuitiv und einfach genutzt werden kann.

3.2 Abfragen mit GraphQL

Für die Entwicklung eines intuitiven und gleichzeitig vielseitigen GraphQL-Schemas ist es wichtig, sich Gedanken über die Geschäftsdomäne des künftigen API-Einsatzgebietes zu machen. Abseits einer einfachen Sprache, die jeder in der Geschäftsdomäne direkt versteht, hilft dabei vor allem, sich darüber bewusst zu werden, wie mit dem API später interagiert wird. Im folgenden Kapitel werden die Interaktionsmöglichkeiten mit einem GraphQL API beleuchtet.

3.2.1 Grundlegende Querys

GraphQL erlaubt es, genau die Information anzufragen, die man benötigt. Als Ergebnis erhält man die Daten in derselben Form wie angefordert zurück. Dafür bietet GraphQL einen Einstiegspunkt, hinter dem sich das komplette zuvor deklarierte Graphenschema aus verwandten Objekten und deren Feldern befindet. Um auf die Daten zuzugreifen, muss ein in der *GraphQL Query Language* definiertes, JSON-ähnliches Objekt mit den Feldern entlang des Graphen an den Endpunkt übergeben werden.

Listing 3–1
Simple GraphQL-Query

```
# Request
{
  product {
    name
}}

# Response
{
  "data": {
    "product": {
      "name": "Kartoffeln"
}}}
```

Wie im Beispiel aufgeführt sendet der GraphQL-Endpunkt ein exaktes Ebenbild des angefragten Objekts zurück; nur in der Antwort als JSON mit den entsprechenden Daten erweitert. Dieselbe Form des Objekts in der Antwort wie im Request zu erhalten, ist ein essenzielles Feature von GraphQL. Hierdurch wird gewährleistet, dass die Response immer in einer erwartbaren Form erfolgt. Dies bedeutet, dass die Antwort des Endpunkts direkt ohne besonderes Mapping in der konsumierenden Applikation die eigenen Objekte mit Werten befüllt.

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 *CustomerService* in einem entsprechenden *service*-Package erstellt.

Listing 8–7
Der Costumer-Typ in schema.graphqls

```
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.

Listing 8–8
Das Customer-Feld gegen den Customer-Service auflösen

```
@Service
public class QueryResolver
    implements GraphQLQueryResolver {

  private CustomerService customerService;

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

```
  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 GraphiQL das `fullname`-Feld des `customer`-Typs aufgerufen werden. Trotz `firstname`- und `lastname`-Feldern in der Datenschicht wird es nun richtig aufgelöst.

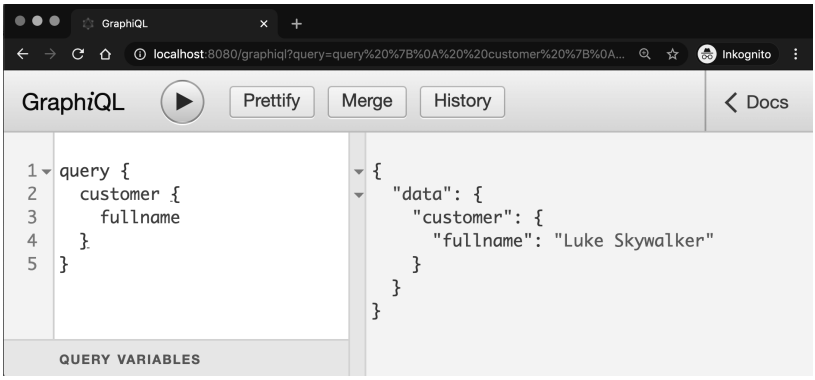


Abb. 8–3
Customer-Fullname-Abfrage mit GraphiQL

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.

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

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

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.

Listing 8–11
Implementieren des Java-Node-Interface

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

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

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 GraphiQL wie in Abbildung 8–4 lösen nun ebenfalls die erzeugte, eindeutige ID des Kunden auf.

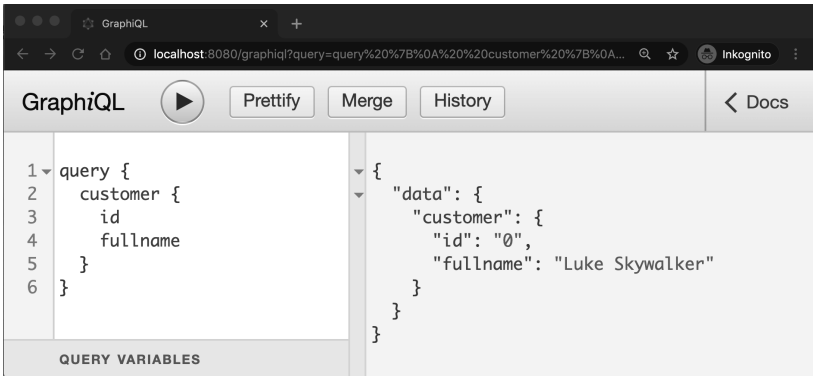


Abb. 8–4
Customer-ID-Abfrage mit GraphiQL

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.

Listing 8–14
Erweiterung des Beispieldatensatzes im CustomerService

```
@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);
  }
}
```

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,
}
```

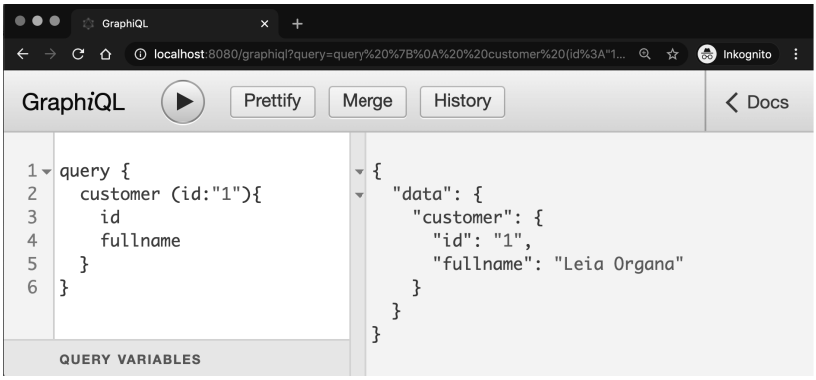
Da es nur einen einzelnen Rückgabewert aus der Liste geben kann, muss der ID-Parameter mit dem Non-Null-Modifikator versehen werden. Um diese Query entsprechend auflösen zu können, erfolgt anschließend die Erweiterung des *QueryResolvers* um den Parameter sowie das korrekte Weiterreichen an den Service.

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 GraphiQL*



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!]!
}
```