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.