I Inhaltsverzeichnis

II Abbildungsverzeichnis

1 Einleitung

Seit jeher ist REST der Standard, wenn es um das Design von http-Schnittstellen geht. Doch mit stetig wachsender Komplexität von Softwaresystemen, ist es um einiges aufwendiger geworden Schnittstellen zu entwerfen oder diese in eigene Anwendungen zu integrieren. Schuld daran ist nicht nur der Dokumentationsaufwand, welcher mit einer größeren API wesentlich höher ausfällt. Weiterhin steigt das Bedürfnis komplexere Systeme und Prozesse mit Hilfe einer Microservicearchitektur abzubilden. Daraus folgt, dass intern weitere Schnittstellen notwendig sind, um einen reibungslosen Betrieb solcher Architekturen zu gewährleisten. Hier kommt der Ansatz von REST-Schnittstellen an seine Grenzen und ebnet den Weg für GraphQL.

Seit GraphQL von Facebook 2012 entwickelt und intern genutzt wurde, ist es seit 2015 Open Source und wird gemeinsam mit deren Community weiterentwickelt.[1] Es erfreut sich an immer größer werdender Beliebtheit. Häufig wird es als Alternative zu REST gesehen. Es wird von Marken wie Audi, PayPal, Twitter, AirBnB und natürlich Facebook selbst genutzt.[2] GraphQL kann das Design von APIs erheblich vereinfachen und somit gerade in komplexen Microservicearchitekturen mit vielen Prozessen und Schnittstellen sein volles Potenzial entfalten. Außerdem vereinfachen unzählige Frameworks wie z.B. Apollo oder GraphQL Mesh den Einsatz von GraphQL selbst. Aus diesen Gründen ist GraphQL auch für die dotSource GmbH von Relevanz. Hier kommt bei der Entwicklung von Webshops in einer Microservicearchitektur immer häufiger GraphQL zum Einsatz, da es die Entwicklung erleichtert und teilweise sogar von Kunden immer mehr forciert wird. Jedoch zeigt GraphQL in diesen Projekten verschiedene Schwächen.

[1] [Byr 15].
[2] [The 22h].

Sollten Bibliotheken wie die von Apollo zum Einsatz kommen, ist es nicht unmittelbar möglich Features wie Federation zu nutzen, um GraphQL-Schnittstellen anzubinden, welche selbst nicht der Spezifikation von Apollo entsprechen.

In dieser Arbeit soll erst geklärt werden, ob externe GraphQL-APIs in eine eigene Microservicearchitektur integriert werden können, in der alle Typen mittels Apollo Federation verteilt sind. Dazu soll die GraphQL-API der Shop Plattform commercetools dienen. Weiterhin soll dabei in diesem Zusammenhang ebenfalls das „N+1“-Problem untersucht werden. Es soll eine Möglichkeit gefunden werden dieses Problem im Rahmen von verteilten Graphen zu lösen.

Dazu wird zunächst die Funktionsweise von GraphQL mit einigen Kernkomponenten erläutert, um ein grundlegendes Verständnis von GraphQL als Basis für diese Arbeit zu schaffen. Danach wird beschrieben, wie GraphQL aktuell in der dotSource GmbH eingesetzt wird. Dabei sollen die bereits kurz erwähnten Probleme genauer betrachtet werden, um die daraus resultierenden Forschungsfragen der Arbeit herauszustellen. Anschließend wird eine Möglichkeit beschrieben diese Probleme in Form eines Microservices zu lösen. Zuletzt wird diese Lösung hinsichtlich ihrer Machbarkeit und Performance evaluiert, um eine Aussage über die Möglichkeit eines Einsatzes in Produktivsystemen treffen zu können.

Diese Arbeit richtet sich demnach vor allem an Systemarchitekten und Entwickler mit grundlegenden Kenntnissen in Software- und Systemarchitekturen.

2 Einführung in GraphQL

GraphQL wird oft fälschlicherweise als Erstsatz zu REST-Schnittstellen gesehen.[3] Doch dabei handelt es sich mehr um eine Abfragesprache für APIs als um ein Architekturkonzept für Netzwerkschnittstellen. [4] GraphQL legt den Fokus viel mehr auf datengetriebenes Design aus Sicht eines Clients und hilft Entwicklern dabei konsistente und selbstdokumentierende Schnittstellen zu implementieren. Daten werden hier nicht als einzelne Ressourcen, welche hinter verschiedene Adressen erreichbar sind, betrachtet. Sie werden viel mehr als Graphen, die sich am Ende an einer einzigen Schnittstelle bündeln, interpretiert. Für diese Graphen wurde eine *query language* (z. dt. Abfragesprache) definiert. Die Graph Query Language bietet also eine neue Möglichkeit Daten an Schnittstellen anzufragen und zu verändern.[5] Dazu nimmt eine GraphQL-Laufzeitumgebung eine Zeichenkette entgegen, welche die entsprechende *Query Language* enthält, um diese Aktionen innerhalb sogenannter *Resolver* auszuführen. Abbildung 1 zeigt schematisch an einem beispielhaften Szenario, wie GraphQL aus verschiedenen Quellen, respektive verschiedenen Graphen, Daten bezieht.

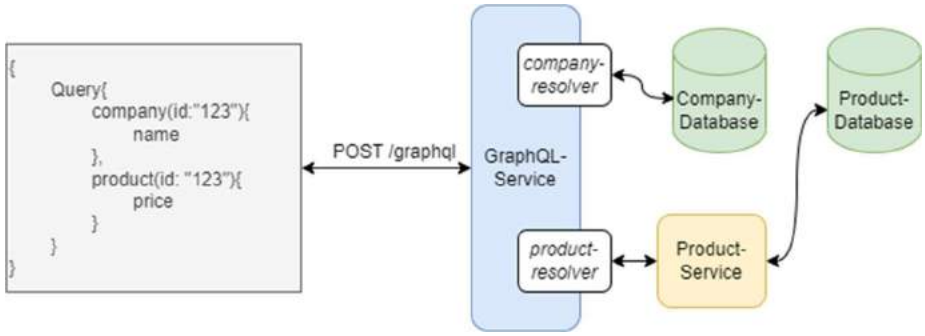


Abbildung 1: GraphQL - Übersicht

Die folgenden Kapitel sollen kurz auf die grundlegenden Konzepte der *Query Language*, der Schemas und der *Resolver* von GraphQL selbst eingehen und anschließend den Unterschied zu REST-Schnittstellen erläutern.

[3] [Stu 17].
[4] [The 22b].
[5] Vgl. ebenda.

2.1 GraphQL Schemas und Typen

Als Basis für alle Operationen mit einer GraphQL Schnittstelle dienen GraphQL Schemas. Für jede Schnittstelle muss genau ein Schema definiert werden, um festzulegen welche Daten dort zur Verfügung stehen und auf welche Art und Weise mit diesen Daten interagiert werden kann.[6] Ein Schema beschreibt also die Struktur der Daten mit all ihren Typen. GraphQL Schemas werden unabhängig von der darunterliegen Technologie in der *GraphQL Schema Language* definiert. Der Aufbau eines solchen Schemas erinnert jedoch dabei an die JavaScript Object Notation (kurz JSON). GraphQL ist streng typisiert und bietet zwei Grundlegende Typen, um Anfragen zu verarbeiten.[7] Zum einen den Typ *Query* (z. dt. Abfrage) um Daten anzufordern und zum anderen *Mutation* (z. dt. Veränderung) um Daten zu erzeugen, zu verändern oder zu löschen.

2.1.1 Der Query-Typ

Obwohl es möglich ist, eine GraphQL-Instanz so zu implementieren, dass eine *Query* Änderungen an Daten vornimmt, ist von den Entwicklern von GraphQL so vorgesehen, dass hier nur Daten angefordert werden, ohne deren Zustand zu ändern. [8] Aus diesem Grund ist es möglich, mehrere Felder innerhalb einer *Query* parallel anzufragen.[9] Da *Query* einer der beiden Root-Typen ist, muss zunächst der Typ *Query* für eine GraphQL-Schnittstelle definiert werden. Abbildung 2 zeigt eine einfache Definition eines solchen Typs zur Abfrage von Firmen und deren Produkten.

```
1  type Query{
2      company(id: String!): Company
3      companies: [Company]
4      product(id: String!): Product
5      products: [Product]
6  }
```

Abbildung 2: Schemadefinition - Query

[6] [The 22d].
[7] Vgl. ebenda.
[8] [The 22f].
[9] Vgl. ebenda.

Eingeleitet wird die Definition mit dem Schlüsselwort *Query*. Dieser Typ besteht wiederum aus den Typen *Company* und *Product.* Dabei kann es sinnvoll sein, für jeden Typen zwei Felder zu definieren. Eines im Singular, um ein einzelnes Objekt eines Typs über dessen ID abzufragen und ein anderes im Plural, um ein Liste von Objekten dieses Typs zu erhalten. Dazu wird jeweils bei *company* und *product* eine ID als Argument in der Parameterliste vorausgesetzt. Das der Parameter vom Typ String erforderlich ist, ist an dem Ausrufezeichen dahinter zu erkennen. Über die Felder *companies* und *products* können jeweils alle vorhandenen Objekte dieses Typs als Liste angefragt werden. Der Rückgabewert ist hier als Liste der jeweiligen Objekte definiert. Parameter, die zu einer möglichen Paginierung dienen könnten, wurden hier weggelassen. Da GraphQL Schemas typisiert sind, bietet es verschiedene Skalare Typen wie *Boolean, String, Int, Float oder ID,* sowie die Möglichkeit eigene Typdefinitionen vorzunehmen, wie es in diesem Beispiel zu sehen ist. Im Folgenden werden die Typen *Company* und *Product* erzeugt. Diese bestehen zunächst aus Attributen mit einem skalaren Typ, können aber auch Attribute mit selbstdefinierten komplexeren Datentypen enthalten.

```
8   type Company {
9       id: ID!
10      name: String
11      catchPhrase: String
12  }
13
14  type Product {
15      id: ID!
16      name: String
17      price: String
18      description: String
19  }
```

Abbildung 3: Schemadefinition - Company & Product

2.1.2 **Der Mutation-Typ**

Im Gegensatz zum *Query*-Typ können Daten mittels *Mutation* verändert werden.[10] Eingeleitet wird die Typdefinition mit dem Schlüsselwort M*utation*, gefolgt von verschiedenen auszuführenden Aktionen. Diese Aktionen lassen sich als eine Art Funktionsaufruf verstehen, an die Parameter übergeben werden können, um Daten zu erzeugen oder zu verändern. Folgender *Mutation*-Typ könnte zum Erstellen und zum Löschen von Produkten dienen.[11]

```
8   type Mutation{
9       addProduct(name: String!, catchPhrase: String): Product!
10      deleteProduct(id: ID!): ID
11  }
```

Abbildung 4: Schemadefinition - Mutation

In diesem Beispiel werden zwei Operationen definiert. Die erste Operation dient zum Hinzufügen von neuen Produkten. Dazu werden die Parameter *name, alter* und *catchPhrase* übergeben. Außerdem ist nach der Parameterliste zusehen, dass diese Operation das erstellte Objekt von Typ *Product* direkt zurückgibt. Die zweite Operation löscht ein Produkt und nimmt dazu die ID des zu löschenden Produktes entgegen. Hierbei handelt es sich ebenfalls um einen Pflichtparameter. Bei Erfolg wird die ID des gelöschten Objektes zurückgegeben. Auch hier kann jede Aktion beliebig oft in einer Anfrage wiederholt werden, um beispielsweise mehrere Produkte innerhalb einer Anfrage hinzuzufügen und zu löschen. Im Gegensatz zu *Queries* werden diese Aktionen jedoch sequenziell ausgeführt.[12]

[10] Vgl. ebenda.
[11] Vgl. ebenda.
[12] Vgl. ebenda.

2.2 Die Resolver

Während Schemas unabhängig von der Technologieumgebung in einer eigens dafür definierten Sprache erstellen werden, müssen sogenannte R*esolver* in einer Programmiersprache implementiert werden, da sie die Logik hinter den Schemas enthalten, um die in den Schemas angegebenen Felder mit Daten zu füllen. Erst die Kombination aus *Schema* und *Resolver* macht einen GraphQL-Server vollfunktional. Für jeden, in einem Schema, definierten Typ in einer *Query* oder für jede Funktion einer *Mutation* muss jeweils ein *Resolver* implementiert werden, um dieses Feld mit Daten zu füllen.[13] Nachdem eine Anfrage an einem GraphQL-Server eingegangen ist, wird deren Schema validiert und anschließend an die entsprechenden *Resolver,* um die geforderten Aktionen auszuführen.[14] Diese *Resolver* enthalten beispielsweise Datenbankabfragen oder Anfragen an andere GraphQL- oder REST-Schnittstellen, um den Datensatz des betreffenden Attributes zu erhalten. Der Rückgabewert des *Resolvers* muss dabei dem Datentyp des Attributes aus dem Schema entsprechen. Anschließend werden alle Attribute herausgefiltert, welche bei der Anfrage nicht explizit angefordert wurden.[15] Anschließend wird das Objekt an den Client zurückgesendet.

Folgender Ausschnitt einer Query dient zur Abfrage eines einzelnen *company*-Objektes und eine Liste von *company*-Objekten.

```
20  type Query {
21    companies(page: Int, size: Int): [Company]
22    company(id: ID): Company
23  }
```

Abbildung 5: Resolver - Ausschnitt einer Query

Beide Felder erwarten zusätzliche Parameter. Bei *companies* können diese Parameter beispielsweis einer Paginierung dienen. Das Feld *company* nimmt eine ID entgegen, um die zugehörigen Daten anzufragen.

---

[13] [The 22a].
[14] Vgl. ebenda.
[15] Vgl. ebenda.

Die beiden *Resolver* dieser *Query* könnten demnach wie folgend aussehen.

```
26  const resolvers = {
27      Query: {
28        company: async (parent, args, context, info) => {
29            return Company.findById(args.id)
30        },
31        companies: const page: any rgs, context, info) =>{
32            const page = args.page ?? 1;
33            const size = args.size ?? 0;
34            return Company.find({}).skip(page*size).limit(size)
35        }
36      },
37  }
```

Abbildung 6: Resolver - Codeausschnitt

In Zeile 26 der Abbildung 6 wird ein Objekt namens *resolver* erzeugt. Diese beinhaltet ein Objekt *Query*, welches die *Resolver*-Funktionen für die Felder *company* und *companies* innerhalb der *Query* aus Abbildung 5 enthält. Ein *Resolver* wird stets mit den vier Parametern *parent, args, context* und *info* aufgerufen.[16] Obwohl in den beiden Funktionen nur die Argumente für die Paginierung und die ID genutzt werden, sind dennoch alle zur Vollständigkeit in der Parameterliste angegeben. Die beiden *Resolver* nutzen die Argumente, um in Zeile 29 und Zeile 34 die entsprechenden Daten abzufragen. Dieses *resolver*-Objekt wird zusammen mit der Schemadefinition dem GraphQL-Server bei der Instanziierung übergeben. Die *Resolver* werden dann vom GraphQL-Server aufgerufen, sollte eine Anfrage eintreffen, in der die Daten des entsprechenden Feldes benötigt werden. So können Daten aus verschiedenen Datenquellen in einem Schema zusammengefasst werden.

[16] Vgl. ebenda.

2.3 Query Language

Nachdem ein Schema und die dazugehörigen Resolver definiert und implementiert wurden, können Anfragen in der *Graph Query Language* in Form einer Zeichenkette an die Schnittstelle gesendet werden.

Hier liegt der größte Vorteil von GraphQL-Schnittstellen. Bei jeder Anfrage ist es erforderlich genau zu definieren, welche Daten in der Antwort benötigt werden. Dieser deklarative Ansatz ermöglicht es den Entwicklern schon bei der Anfrage die Datenstruktur der Antwort zu formen. [17] So ist die Antwort leicht vorhersehbar und es ist einfacher diese Schnittstelle an andere Systeme anzubinden. Folgendes Beispiel zeigt eine passende Abfrage zum Schema aus Kapitel 2.1.1.

```
query {
    company(id: 1){
        id
        name
    }
    product(id: 1){
        price
    }
}
```

Abbildung 7: Query Language - Query

In dieser Anfrage werden eine Firma und ein Produkt angefordert. Jedoch ist nur der Preis des Produktes, der Name der Firma interessant. In dieser Anfrage werden verschiedene Ressourcen, welche möglicherweise intern aus unterschiedlichen Systemen oder Datenbanken stammen, angefordert. Weiterhin werden dank des deklarativen Ansatzes von GraphQL nur die Informationen aus diesen Datensätzen ausgewählt und zurückgegeben die bei der Anfrage angefordert werden.

[17] Vgl. ebenda.

Betrachtet man nun die folgende dazugehörige Antwort fällt auf, dass sich die Strukturen ähnlich sind.

Abbildung 8: Query Language – Query Antwort

In Abbildung 7 wurden Firma und Produkt über ihre ID ausgewählt, da die Typen der *Query* die Angabe dieser Parameter erforderte. Die Antwort erhält dann ausschließlich die angeforderten Daten anstelle des gesamten Objektes. Wie bereits in Kapitel 2.1 erwähnt, ist es auch möglich ganze Listen von Objekten eines Typs anzufordern, sollte das Schema es unterschützen. So könnten ausschließlich die Preise aller Produkte wie folgt angefragt werden.

Abbildung 9: Query Language - Anfragen einer Liste

Dazu wird das Feld *products* genutzt und ausschließlich das *price*-Attribut angegeben. Die Antwort enthält dann ausschließlich eine Liste der Preise aller Produkte.

Abbildung 10: Query Language - Antwort auf einer Listenanfrage

Es möglich mehrere *Queries* auszuführen, in dem sie untereinander in einer Anfrage gebündelt werden.[18]

Abbildung 11: Query Language - Parallele Queries

Bei *Mutations* ist es ebenfalls möglich beliebig viele *Mutations* in einer Anfrage zu bündeln, um verschiedene Daten zu verändern. *Query* und *Mutation* sind dabei jedoch nicht gleichzeitig in einer Anfrage verwendbar.

[18] [Apo 22d].

Weiterhin können bei *Mutations* ebenfalls die für die Antwort relevanten Attribute angegeben werden, um den veränderten Zustand zu prüfen. Eine *Mutation* sieht dabei wie folgt aus.

```
mutation{
    addproduct(name: “Produktname“, price: 19.99){
        id
        name
        price
    }
}
```

Abbildung 12: Query Language - Mutation

Wie der Name des Attributes *addProduct* vermuten lässt, kann hier ein neues Produkt mit den übergebenen Parametern angelegt werden. In der Antwort werden direkt der Name, die ID und der Preis des neu erzeugten Objektes ausgegeben, um mit den neuen Daten direkt weiterarbeiten zu können.

```
{
    "data":{
        "addProduct":{
        "id": 123,
        "name": "Produktname",
        "price": 19.99
        }
    },
    "errors":[]
}
```

Abbildung 13: Query Language - Mutation Antwort

2.3.1 Introspection und GraphiQL

Ein weiterer großer Vorteil von GraphQL ist das *Introspection*-System. Es bietet die Möglichkeit Informationen über das von der Schnittstelle unterschützte Schema selbst abzufragen.[19] Dazu muss der Typ *__schema* unter Angabe der relevanten Attribute angegeben werden.[20] Folgende Anfrage gibt Auskunft über alle unterstützen Typen der GraphQL-Schnittstelle des obigen Beispiels.

```
{
    __schema {
        Types{
            Name
        }
    }
}
```

Abbildung 14: Introspection - Anfrage

Auf diese Weise gibt die Schnittstelle die Namen aller zur Verfügung stehenden Typen aus.[21] Es ist auch möglich neben dem Namen auch weitere Informationen zu verschiedenen Typen abzufragen. Der doppelt Unterstrich ist dabei ein Zeichen dafür, dass es sich bei dem Feld um ein Feature von *Introspection* handelt.

```
{
    "data": {
    "__schema":{
        "Types":[{
            "name": "Product"
        },
        {
            "name": "Company"
        },
        ...
        ]
    }
    },
    "errors":[]
}
```

Abbildung 15: Introspection - Antwort

[19] [The 22e].
[20] Vgl. ebenda.
[21] Vgl. ebenda.

In der Antwort werden unter anderem die Typen *Company* und *Product* ausgegeben. Die in *Company* und *Product* verschachtelten Typen wurden hier aus Gründen der Übersichtlichkeit gekürzt, können aber ebenfalls eingesehen werden. Weiterhin können zusätzliche Informationen wie z.B. *kind* oder *field* über die Typen abgefragt werden, um Inforationen über die Art des Typs und dessen Attribute zu bekommen.[22] Jedoch sind für alle diese Informationen erst Anfragen an die Schnittstelle notwendig. Hier geht GraphQL jedoch noch einen Schritt weiter und bietet abhängig von der genutzten Bibliothek oftmals eine Oberfläche namens *GraphiQL* an. Hierbei handelt es sich im eine Web-Oberfläche. Dabei übernimmt die Web-Oberflächen alle nötigen Anfragen, um mittels der *Introspection* alle Verfügbaren Typen und deren Eigenschaften übersichtlich anzuzeigen.

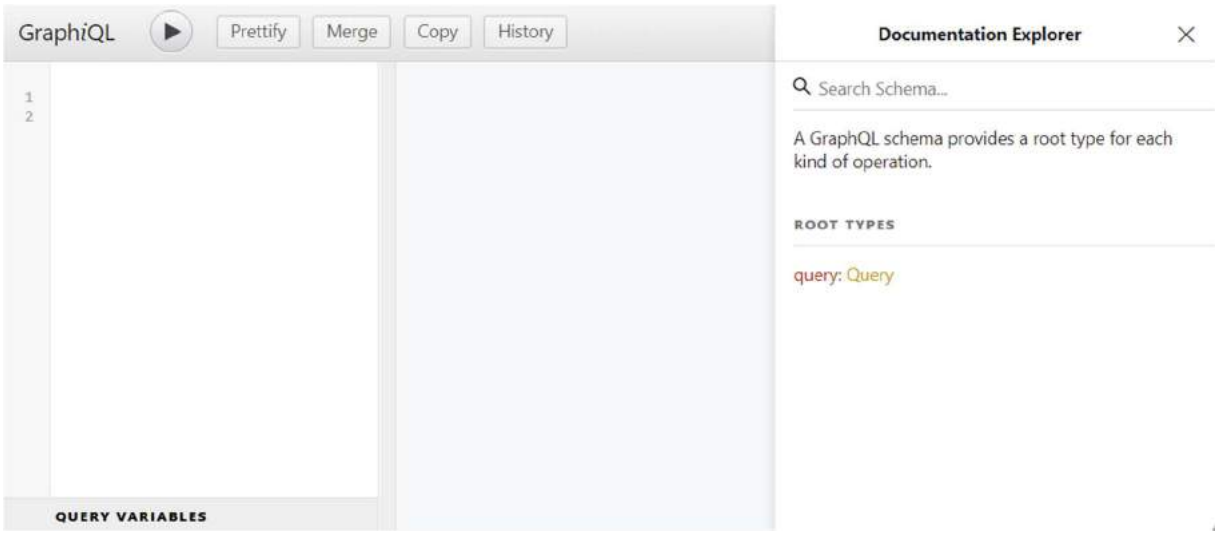


Abbildung 16: GraphiQL – Aufbau I

[22] Vgl. ebenda.

Auf der linken Seite bietet die Oberfläche einen Bereich, um GraphQL-Anfragen zu formulieren und diese anschließend zu versenden. Dank es bereits ermittelten Schemas gibt es hier eine Autovervollständigung beim Schreiben von Anfragen. Die Antwort ist dann im mittleren Bereich zu sehen. Rechts bietet die Oberfläche den *Documentation-Explorer,* um das zur Verfügung stehende Schema anzuzeigen und direkt korrekte Anfragen daraus zu formulieren.

Abbildung 17: GraphiQL - Aufbau II

Dieses Werkzeug ermöglich es Entwicklern die Schnittstellen einfach zu verstehen und zu testen. Zugleich liefert sich eine Dokumentation aller Verfügbaren APIs.

2.4 Das Apollo-Framework

Wie in vorherigen Kapiteln bereits erwähnt, handelt es sich bei GraphQL nicht etwa um ein Architekturkonzept für Netzwerkschnittstellen, sondern vielmehr um eine Abfragesprache und zahlreiche Werkzeuge um Netzwerkschnittstellen effizienter zu entwerfen und zu verwenden. Dabei ist Apollo ein weit verbreitetes und beliebtes Framework. Wie die Nutzerzahlen einiger Bibliotheken von Apollo zeigen. Apollo bringt neben einer eigenen GraphQL-Laufzeitumgebung mit etwas erweiterter Funktionalität weitere Werkzeuge rund um GraphQL-Schnittstellen mit.[23] Beispielweise erleichtert Apollo das Anbinden solcher Schnittstellen mit einem eigenen Apollo-Client für Android, IOS und React, um mobile Anwendungen sowie Webseiten effizient an einen Apollo-Server anzubinden. Weiterhin bietet es eine Web-Oberfläche ähnlich zu GraphiQL aus Kapitel 2.3.1 namens Apollo Studio.

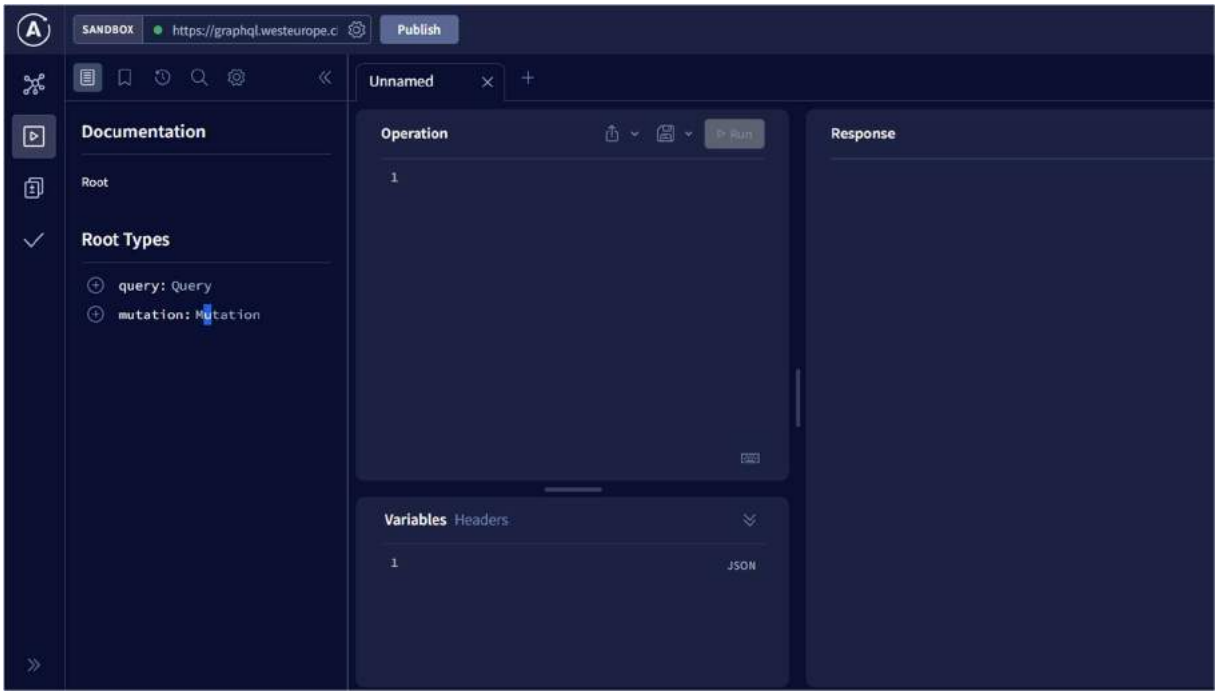


Abbildung 18: Apollo Studio - Übersicht

[23] [Apo 22a].

In Abbildung 18 ist eine ähnliche Struktur zu in GraphiQL zusehen. Dieses Mal wird das via *Introspection* ermittelte Schema im linken Drittel der Seite dargestellt. Im mittleren Bereich kann die Anfrage ebenfalls mit Autovervollständigung formuliert werden. Es gibt jedoch einen großen Unterschied zu GraphiQL. Apollo Studio ist im Gegensatz zu GraphiQL eine cloudbasierte Plattform.[24] Die in Abbildung 18 gezeigte Oberfläche stammt aus dieser Cloud. Sie kann unter *https://studio.apollographql.com/sandbox/explorer* aufgerufen werden. Unter Angabe der Adresse, der zu testenden API im Feld in der oberen linken Ecke, kann eine eigene API getestet werden. So wird das Schema von der angegeben Adresse abgerufen und alle Anfragen dorthin gesendet. Auch bei Aufruf einer eigens implementierten GraphQL-API, wird zunächst auf diese Seite weitergeleitet und diese entsprechende Adresse automatisch in dieses Feld eingetragen.

In dieser Plattform stellt Apollo außerdem einige weitere Features, neben Federation zur Verfügung. In dieser Arbeit sollen sie jedoch nicht weiter betrachtet werden sollen. Bei Federation handelt es dich um die Kernkomponente des Apollo-Frameworks, um mehrere Schemas an einer Stelle zusammenzufassen. Im Folgenden Kapitel soll die Funktionsweise von Federation geklärt werden.

[24] [Apo 22g].

2.4.1 Apollo Federation

Da GraphQL allem Anschein nach immer häufiger in Microservices eingesetzt wird, entstehen viele kleinere Schemas für jeden Service. Besonders in der dotSource GmbH sind Microservices so implementiert, dass jeder Service für einen einzigen Datentyp verantwortlich ist. So kann er daher nur den Datentyp in seinem Schema beschreiben, für den er zuständig ist. Auf diese Weise entsteht eine Vielzahl einzelner GraphQL-Services mit verteilen Schemas innerhalb der Architektur. Im Kontext von Apollo werden diese auch Subgraphen genannt.[25] Um Clients zentral an diese Struktur aus Subgraphen anbinden zu können, müssen diese an einem Gateway gebündelt und die Schemas zusammengefasst werden. Solch ein gebündeltes Schema wird auch Supergraph genannt.[26] Hier entsteht zusätzlich die Herausforderung, dass sich die Subgraphen untereinander referenzieren können, wenn ein Objekt eine ID eines anderen Objektes als Sekundärschlüssel enthält. Es ist zudem möglich, dass diese Subgraphen ihre Daten aus unterschiedlichen Quellen beziehen und Querverweise nur schwer aufzulösen sind. Folgende Graphen sind dabei denkbar:

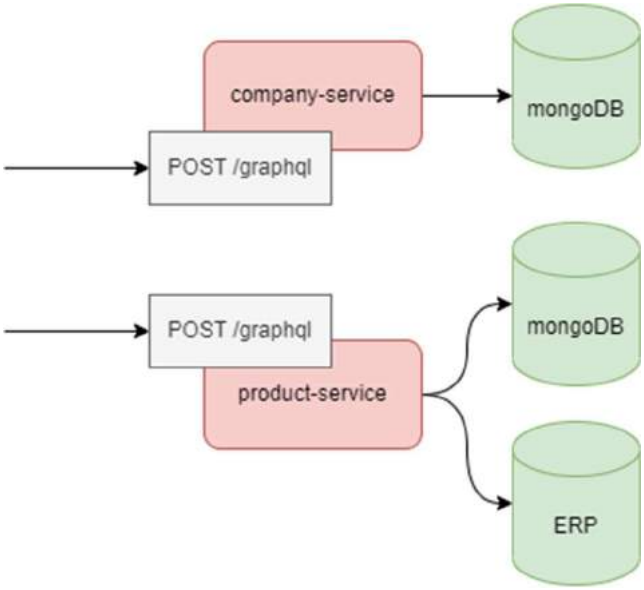


Abbildung 19: Apollo Federation - Subgraphen

[25] [Apo 22e].
[26] [Apo 22f].

Die Abbildung 19 zeigt schematisch den Aufbau zwei solcher Subgraphen. Ein *company*-service stellt Informationen zu einer Firma bereit und bezieht dafür die Daten aus einer Datenbank. Weiterhin ist der *product*-service für Produktdaten zuständig und bezieht einen Teil der Daten aus einer anderen Datenbank und einen Teil aus einem Enterprise Resource Planning System. Außerdem ist in der Produktdatenbank die ID der Firma gespeichert, welches das Produkt hergestellt hat. Sonst sind in den Graphen keine weiteren Informationen des jeweils anderen enthalten.

Nun sollen Produktdaten vom *product*-service abgefragt, der dazugehörige Hersteller aufgelöst und dessen Daten ebenfalls ausgegeben werden. In Kapitel 2.1.1 wurde in Abbildung 3 ein Ausschnitt der Schemadefinition für Produkte und Firmen beschrieben. Gibt es für jeden Typ einen einzelnen Service mit eigenem Schema. Um diese Schemas in einen Subgraphen zu verwandeln, müssen leichte Anpassungen an den Typdefinitionen gemacht werden.

```
1  type Query{
2      company(id: String!): Company
3      companies: [Company]
4  }
5
6  type Company @key(fields: "id") {
7      id: ID!
8      name: String
9      catchPhrase: String
10 }
```

Abbildung 20: Apollo Federation - Schemadefinition I

Abbildung 20 zeigt das Schema des *company*-service. Der Typ *Company* muss um eine Direktive namens *key* erweitert werden, um es zu einer sogenannten Entität zu machen. [27] Apollo Federation kann laut Spezifikation nur mit solchen Entitäten arbeiten. Mit dieser Direktive kann angegeben werden, welches Feld ein Objekt dieses Typs eindeutig identifiziert. Die Definition des *product*-service zeigt den nächsten notwendigen Schritt.[28]

[27] [Apo 22b].
[28] Vgl. ebenda.

```
1   type Query {
2       products: [Product]
3       product(id: ID): Product
4   }
5
6   type Product @key(fields: "id") {
7       id: ID!
8       name: String
9       price: String
10      description: String
11      company: Company
12  }
13
14  extend type Company @key(fields: "id") {
15      id: ID! @federation__external
16  }
```

Abbildung 21: Apollo Federation - Schemadefinition II

Abbildung 21 zeigt das Schema des *product*-service. Auch hier fällt sofort auf, dass der Typ *Product* mit der *key*-Direktive erweitert wurde, um es zu einer Entität zu machen. In Zeile 11 kann jetzt der *Product*-Typ um ein Feld vom Typ *Company* erweitert werden, obwohl dieser Typ in einem völlig anderen Service definiert ist. Dazu muss lediglich der Typ *Company* mit dem Schlüsselwort *extend* angegeben werden. Somit gibt dieser Subgraph nach außen bekannt, dass dieser Typ an einer anderen Stelle definiert ist. Mit der Direktive *federation_external* wird angegeben, dass die ID ebenfalls extern bereitgestellt wird, um diesen Typen zu dereferenzieren.[29]

Zuletzt muss im *company*-service eine zusätzliche *Resolver*-Funktion implementiert werden.

```
Company: {
  __resolveReference(ref){
    return Company.findById(ref.id)
  }
},
```

Abbildung 22: Apollo Federation - Resolver

[29] [Apo 22c].

Dieser *Resolver* muss zusätzlich zu den bereits vorhandenen *Resolvern* aus Kapitel 2.2 implementiert werden. Laut Spezifikation muss der Name der Funktion *__resolveReference* lauten.[30] Diese Funktion wird mit den üblichen Parametern für *Resolver* aufgerufen, wurden hier zur Übersichtlichkeit jedoch ausgelassen, da sie in der Funktion nicht benötigt werden. Nur der erste Parameter ist hier von Bedeutung. Er enthält das Eltern-Objekt. Dieses Objekt ist in diesem Fall ein Objekt des Typen *Product,* da ein *Product* laut Abbildung 21 nun ein *Company*-Objekt enthalten kann. In dieser Abbildung ist in Zeile 7 und durch die Direktive festgelegt wurden, dass dieses Objekt über eine ID eindeutig zu identifizieren ist. Also kann es mit *ref.id* von einer Datenbank (in diesem Fall eine MongoDB) angefragt werden.
Zuletzt müssen diese Subgraphen zu einem Supergraphen zusammengefasst werden. Dieser Supergraph wird von einem Apollo-Gateway genutzt und die APIs der beiden Services sind zentral an dieser Stelle verfügbar. Dieses Gateway sorgt anschließend dafür das alle Typen der verschiedenen Schnittstellen angefragt werden können. Da diese sich untereinander referenzieren, löst das Gateway die Referenzen entsprechen auf und stellt die Typen verschachtelt zur Verfügung. So ist beispielsweise folgende *Query* problemlos möglich. In Abbildung 23 kann direkt innerhalb eines Produktes die Firma abgerufen werden, die es produziert hat.

Abbildung 23: Apollo Federation - Query mit Referenz

```
1  query {
2      product(id: "123"){
3          name
4          price
5          company {
6              name
7          }
8      }
9  }
```

[30] [Apo 22b].

Das Gesamtbild dieses Aufbaus stellt sich dann wie folgend dar.

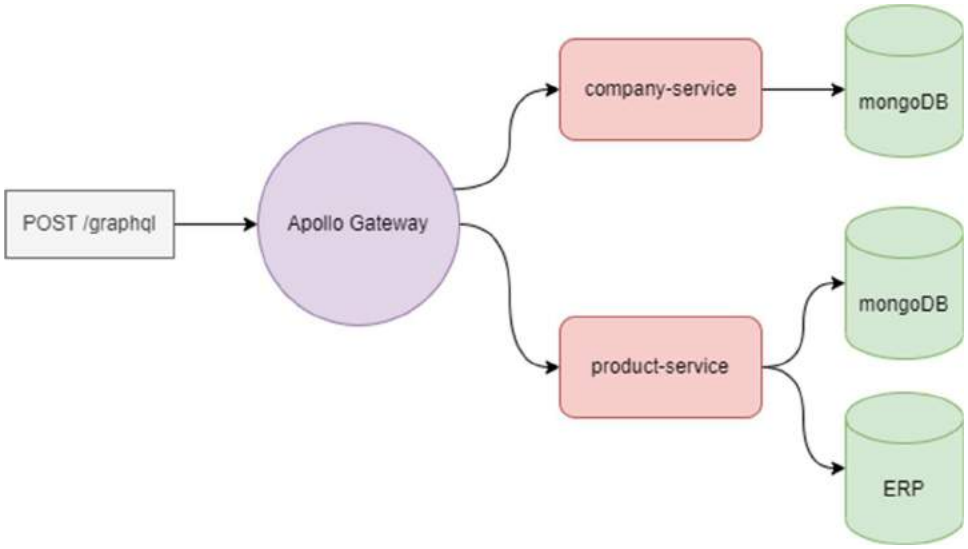


Abbildung 24: Apollo Federation - Supergraph

Nun kann über den Supergraphen des Apollo-Gateway jeder einzelne Typ aus verschiedenen Datenquellen angefragt werden. Das Schema sieht in Bezug auf das vorangegangene Beispiel dabei folgendermaßen aus.

```
type Query {
    products: [Product]
    product(id: ID): Product
    companies: [Company]
    company(id: ID): Company
}
```

Abbildung 25: Apollo Federation - Schema des Supergraphes

2.5 Unterschiede zu REST-Schnittstellen

2.5.1 Ressourcen und Methoden

Obwohl es sich bei GraphQL also mehr um ein Framework als um ein Netzwerkarchitekturkonzept handelt, wird es dennoch oft als Alternative zu REST-Schnittstellen gesehen. Zunächst unterscheiden sich beide Ansätze in ihrer Art und Weise Ressourcen zu strukturieren. Eine REST-Schnittstelle stellt jede Ressource unter einem bestimmten Pfad, dem sogenannten *Uniform Resource Identifier*, zur Verfügung.[31] Auf diesen Pfaden können dann, mittels der verschiedenen im Standard RFC 7231 definierten http-Methoden, verschiedene Aktionen ausgeführt werden. Dabei wird z.B. *GET* zum Abfragen, *POST* zum Erstellen, *PUT* zum Verändern und *DELETE* zum Löschen der Ressource genutzt.[32] GraphQL hingegen kommt mit einer einzigen Adresse aus, um alle verfügbaren Ressourcen bereitzustellen. Weiterhin wird in den meisten Fällen ebenfalls nur eine einzige http-Methode genutzt, um mit der Schnittstelle zu kommunizieren.[33] Der Client sendet nur *POST*-Anfragen an diese eine GraphQL-Schnittstelle, welche in dem meisten Fällen unter dem Pfad */graphql* definiert ist. Diese Anfragen enthalten im *Body* die Befehle in Form der durch GraphQL definierte Abfragesprache, um unterschiedliche Ressourcen zu erhalten.

[31] [Jak 05].
[32] [Fielding et al. 04a].
[33] [The 22c].

Folgendes Beispiel zeigt eine einfache Abfrage von drei Autoren über ihre ID. Soll dafür, wie rechts im Bild, eine REST-Schnittstelle genutzt werden, sind drei *GET*-Anfragen nötig, in denen jeweils die ID des gewünschten Autors separat als Parameter übergeben wird. Wird stattdessen eine GraphQL-Schnittstelle genutzt, können die gewünschten Autoren innerhalb einer *POST*-Anfrage angefordert werden. Dazu werden die Objekte mit ihrer ID in der GraphQL *Query Language* angegeben und es wird mit einem Objekt, welches die Datensätze der Autoren enthält, geantwortet.

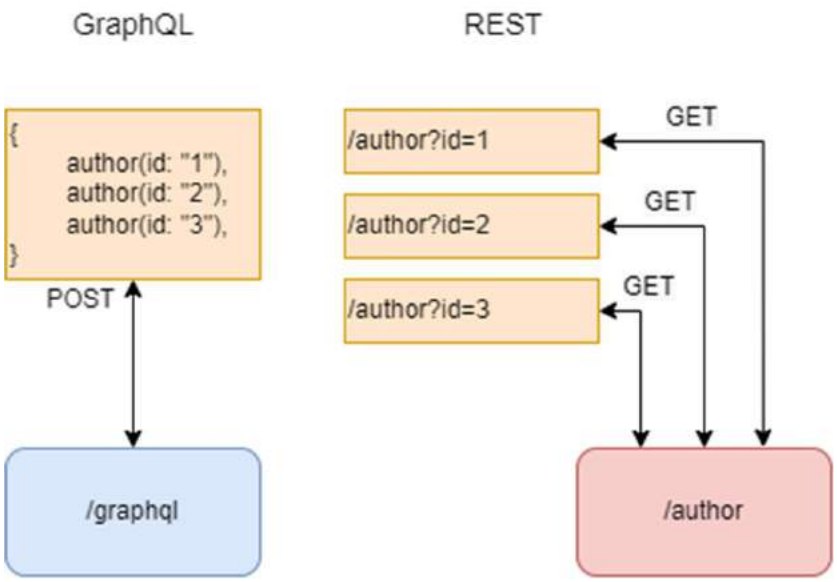


Abbildung 26: GraphQL vs. REST

2.5.2 **Fehlerhandling**

Das Fehlerhandling unterscheidet sich gänzlich zwischen diesen beiden Ansätzen. Eine REST-Schnittstelle quittiert eine angeforderte Aktion immer mit dem passenden Statuscode. Wurde die betreffende Aktion erfolgreich ausgeführt wird etwa mit *20x,* bei einer falschen Anfrage mit *40x* und bei einem internen Fehler auf Seite des Servers mit einem 50x Status geantwortet.[34] Abhängig davon ändert sich die Struktur des zurückgegebenen Objektes. Während also bei einer erfolgreichen Anfrage immer das gleiche statische Objekt zurückgegeben wird, kann im Falle eines Fehlers nur ein Fehlerobjekt mit einem Fehlercode und ggf. einer Nachricht zurückgeben werden. Da diese Objekte in diesem Fall nicht klar definiert sind, kann sich beispielsweise die Struktur der Fehlerobjekte von Schnittstellt zu Schnittstelle ein wenig ändern. Als Entwickler einer REST-Schnittstelle muss also jedes Mal darauf geachtet werden, dass die Schnittstellen verschiedener Ressourcen, hinsichtlich der zurückgegebenen Statuscodes und Objekten ähnlich sind oder am besten auf die gleiche Art und Weise funktionieren, um das Anbinden dieser Schnittstellen für Dritte zu erleichtern.
GraphQL erleichtert es hier einheitliche Schnittstellen zu entwickeln. Fehler werden nicht mit Hilfe eines Statuscodes abgebildet, sondern über ein in der Antwort befindliches Attribut. Daher hat jede Antwort eines GraphQL-Servers immer die gleiche Struktur. Zurückgegeben wird immer ein Objekt derselben Struktur, mit den Attributen *data* und *error*. [35] Das *data*-Attribut enthält das Ergebnis der gewünschten Aktionen, während *error* eine Liste aller Fehlerobjekte enthält, die beim Durchführen der Aktionen aufgetreten sind. Da sich der Statuscode sowie die Struktur der Antwort nie ändert, ist die Entwicklung und Anbindung solcher Schnittstellen weniger komplex.

[34] [Fielding et al. 04b].
[35] [The 22g].

2.5.3 **Caching**

Im Konzept von REST-Schnittstellen ist Caching als eine Kernkomponente fest verankert.[36] Es baut auf den Spezifikationen des http-Standards auf und bezieht diese direkt in das Caching mit ein. So werden verschiedene URIs und Methoden genutzt, um mit unterschiedlichen Ressourcen zu interagieren. Dadurch ist es möglich eine Anfrage eindeutig zu identifizieren und deren Ergebnis zwischenzuspeichern. Betrachten wir dazu folgendes Beispiel. Ein Client möchte alle Daten zu einer bestimmten Firma abfragen. Dazu löst er folgende *GET*-Anfrage aus.

```
GET /company?id=123456
```

Als Antwort wird der Datensatz der betreffenden Firma zurückgegeben. Nun lässt daraus ein Cache-Eintrag bilden, indem alle oben genannten Informationen der Anfrage, also der Pfad, die http-Methode und ggf. die zusätzlichen Parameter als Schlüssel genutzt werden, um diese Anfrage eindeutig zu identifizieren. Da REST-Schnittstellen zustandslos sind und bei gleichen Anfragen immer gleiche Antworten liefern[37], kann diese Antwort nun in Kombination mit dem Schlüssel in einen Cache gespeichert werden. Sollte die gleiche Anfrage noch einmal gestellt werden, kann das Ergebnis viel schneller aus dem Cache geholt werden.

Da GraphQL hier nicht den http-Standards folgt, sondern http viel mehr nur als Plattform nutzt, um dort eigene Konzepte umzusetzen, ist Caching auf Basis der http-Methode und der URI nicht mehr möglich. Alle GraphQL-Anfragen gehen immer via *POST* auf den gleichen Pfad und sind somit von außen nicht mehr zu unterscheiden. Das Caching muss vom Client selbst oder direkt auf dem Server vorgenommen werden, da dazu nur noch der *Body* der Anfrage als Merkmal genutzt werden kann, um Unterscheidungen vornehmen zu können. Bei verschlüsselten Verbindungen haben ausschließlich des Clients und der Server Zugriff auf die Daten einer Anfrage. Zunächst kann ein Entwickler einer API sich nicht darauf verlassen, dass fremde Clients ein lokales Caching implementieren, um seine Schnittstellen nicht zu

[36] [Jak 05].
[37] Vgl. ebenda.

überlasten. Infolgedessen könnte ein serverseitiges Caching hinter der Schnittstelle implementiert werden, um möglicherweise für Entlastung zu sorgen. Im schlimmsten Fall die Performance nur geringfügig verbessert, da der Client zum einen die Anfragen zunächst trotzdem versenden muss und zum anderen die Anfragen an der Schnittstelle dennoch erst einmal entgegengenommen werden müssen. Zusammenfassend bleibt zu sagen, dass Caching in einem GraphQL-Umfeld am besten direkt clientseitig implementiert werden sollte, wenn sichergestellt werden kann, dass nur bekannte Clients mit den Schnittstellen kommunizieren. Caching lässt sich vermutlich also effizienter in einer Umgebung mit REST-Schnittstellen umsetzten. Dies müsste aber weiteren Evaluierungen unterzogen werden.

2.5.4 **Over-fetching und under-fetching**

Der größte Unterschied zu REST liegt jedoch in den Möglichkeiten, wie Daten abgefragt oder verändert werden können. GraphQL löst hier eines der Probleme, die REST-Schnittstellen mit sich bringen. Gemeint ist hier das sogenannte *over-fetching* bzw. *under-fetching*.

Bei *over-fetching* handelt es sich um Anfragen, bei denen die Antwort mehr Informationen enthält als tatsächlich benötigt werden.[38] Wenn beispielsweise eine Datenbank mit Informationen zu Produkten über eine REST-Schnittstelle angefragt wird, um lediglich die Namen aller Firmen zu erhalten, muss dennoch jeweils der komplette Datensatz über die Firma zurückgegeben werden. Da REST-Schnittstellen meistens statische Datenstrukturen zurückgeben, enthalten sie oft für die Anwendung unnötige Daten. Dadurch erhöht sich die zu übertragende Datenmenge und der benötigte Arbeitsspeicher, um die Antwort programmatisch auszuwerten. Bei einer GraphQL-Anfrage hingegen, kann genau definiert werden, welche Attribute aus den Datensätzen benötigt werden. [39] Wird innerhalb des Firmen-Objektes nur das *name*-Attribute angegeben, so enthält das Objekt in der Antwort auch nur den Namen. Es wird hier also zugunsten der Performance und Speichergröße der Antwort nur eine Liste mit Namen ausgegeben. Das Folgendes Beispiel zeigt eine solche Anfrage und deren Antwort.

```
1  query {
2      companies{
3          name
4      }
5  }
```

Abbildung 27: GraphQL - Over-fetching

[38] [Mukhiya et al. 19], S. 339.
[39] [The 22f].

Hier wird von der Ressource *companies* nur der Name angefragt. Typischerweise kann jede GraphQL-Ressource im Singular für eine bestimmtes Objekt oder im Plural für eine Liste aller Objekte des gleichen Typs angefragt werden. Eine mögliche Antwort könnte demnach wie folgt aussehen. Sie enthält zu jedem *company*-Objekt jeweils nur den Namen.

Abbildung 28: GraphQL - Over-fetching Antwort

Beim *under-fetching* ist genau das Gegenteil der Fall. Bietet eine REST-Schnittstelle nicht genügend Informationen, muss ggf. eine weitere Schnittstelle angefragt werden, um alle benötigten Informationen zu erhalten.[40] Da hier nun gleich mehrere Schnittstellen angefragt werden, welche ebenfalls ihre statischen Datenstrukturen zurückgeben, spielt auch hier wieder *over*-fetching eine große Rolle, da sich die Menge der unnötig übertragenden Daten mit jeder zusätzlich benötigten Schnittstelle potenziell erhöht. GraphQL ermöglicht es gleich mehrere Ressourcen in einer Anfrage anzufordern und auch hier wieder nur die nötigen Attribute auszuwählen um *over-fetching* zu vermeiden. Folgendes Beispiel soll GraphQL-Anfragen noch einmal verdeutlichen.

[40] [Mukhiya et al. 19], S. 339.

```
query {
    comapny(id: "123"){
        name
    },
    product(name: "Produkt"){
        comapny{
            name
        },
        price
    }
}
```

Abbildung 29: GraphQL - Under-fetching Anfrage

Hier werden zwei Ressourcen parallel angefragt, wo bei REST-Schnittstellen zwei einzelne Anfragen nötig wären. Es soll neben dem Namen der Firma mit der ID „123“ auch ein Produkt mit dem Namen „Produkt“ zurückgeben werden. Dabei soll ebenfalls der Name der Firma, sowie der Preis des Produktes ausgegeben werden. Eine mögliche Antwort könnte wie folgt aussehen:

```
{
    "data": {
        "company":{
            "name": "Firmenname"
        },
        "product": {
            "comapny":{
                "name": "Firmenname"
            },
            "price": 500.00
        }
    },
    "errors":[]
}
```

Abbildung 30: GraphQL - Under-fetching Antwort

Diese Antwort enthält nur die tatsächlich benötigten Attribute der einzelnen Ressourcen und umgeht das Problem des *over-fetching*. Ebenfalls fällt auf, dass sich die Anfragen und die Antworten in ihrer Struktur gleichen. Dies ist ein weiterer Vorteil von GraphQL-Schnittstellen, da jeder Entwickler genau die Antwort bekommt, die auch erwartet wird.

3 Die Probleme von GraphQL

In diesem Kapitel soll der Einsatz von GraphQL in einer Microservicearchitektur behandelt werden. Dabei sollen zunächst Einsatzmöglichkeiten und verschiedene Ansätze, GraphQL in solch eine Umgebung zu implementieren, aufgezeigt werden. Weiterhin wird beschrieben, wie GraphQL in verschiedenen Kundenprojekten der dotSource GmbH eingesetzt wird, um anschließend einige daraus resultierende Probleme aufzuzeigen.

3.1 Aktueller Einsatz von GraphQL

Die dotSource GmbH betreut ihre Kunden in verschiedenen Themen hinsichtlich der voranschreitenden Digitalisierung, unter anderem auch bei der Modernisierung oder dem Neuaufbau von Onlineshops. Dabei kommen neben den unterschiedlichsten Technologien von SAP, Salesforce und Intershop auch die Webshopplattform der commercetools GmbH zum Einsatz. Bei commercetools handelt es sich um einen Software-as-a-Service Anbieter, um für einen Onlineshop wichtige Prozesse und Ressourcen abzubilden. Dazu werden eine Vielzahl von REST-APIs angeboten, um Kundendaten, Warenkörbe, Bestellungen und Preise zu verwalten und Prozesse wie z.B. eine Nutzeranmeldung oder das Bestellen von Warenkörben in einem Onlineshop durchführen zu können. Neben der Haltung wichtiger Geschäftsdaten stellt commercetools auch fundamentale Prozesse über verschiedene APIs zur Verfügung, welche mit einem Frontend verbunden werden können, um einen sehr einfachen Onlineshop mit grundlegenden Funktionen bereitzustellen. Da es sich jedoch meistens um größere Kunden handelt und diese schon einige bei sich etablierte Prozesse und andere Systeme, wie z.B. ERP-Systeme oder PIM-Systeme mitbringen, ist es nicht ausreichend, ein Frontend direkt mit der API von commercetools zu verbinden. Außerdem bringen Kunden meistens individuelle Wünsche nach Features mit, die in ihren neuen Onlineshops verfügbar sein sollen, welche nicht mit der Standardfunktionalität von commercetools abzubilden sind. Um diese Prozesse korrekt abzubilden, wird eine Microservicearchitektur als Verbindung zwischen commercetools und einem Frontend implementiert, um zusätzliche Datenbanken und Systeme anzubinden und diese in individuelle Prozesse zu integrieren. Daraus ergibt sich ein klassisches 3 Schichtenmodell.

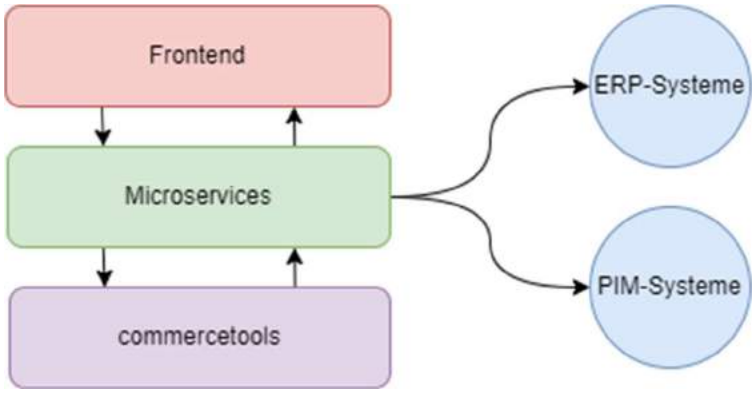


Abbildung 31: 3 Schichtenmodell - Überblick

Des Weiteren bietet commercetools seit 2018 eine GraphQL-API an. Seit 2020 wird diese GraphQL-API auch in der dotSource GmbH vermehrt als Alternative zu deren REST-APIs genutzt. Diese GraphQL-API wurde zunächst nur vereinzelt in einigen Kundenprojekten genutzt, da die gesamte Architektur auf REST-APIs ausgelegt war. Beispielsweise ist es in einigen Projekten notwendig gewesen, Rechnungs- und Versandadressen gesondert zu behandeln und dafür in eigene Microservices auszulagern. Jedoch bietet commercetools keine separate REST-API, um Adressobjekte zu verwalten. Stattdessen sind die Adressen Teil des viel größeren Kundenobjektes, welches komplett über die REST-API der Kunden angefragt werden muss, um anschließend die Adressen herauszufiltern und auszugeben. Hier konnte die GraphQL-API sehr vorteilhaft genutzt werden, um ausschließlich das Adressobjekt bestimmter Kunden auszugeben.

Dazu wurde wie folgt eine statische Anfrage implementiert.

```
11  function createBillingAddressQueryString(customerId) {
12      return `query { customer(id: "${customerId}"){
13          billingAddresses {
14              id
15              company
16              department
17              firstName
18              lastName
19              streetName
20              streetNumber
21              additionalStreetInfo
22              postalCode
23              city
24              country
25              externalId
26              phone
27          }
28      }
29  }`;
30  }
```

Abbildung 32: Ist-Stand - Statische Query

Diese in Abbildung 32 dargestellte Zeichenkette wird jedes Mal vollständig an commercetools gesendet, um die Rechnungsadresse eines Kunden zu erhalten. Dabei ist lediglich die *customerId* in Zeile 12 variabel, um die korrekten Kundendaten abzufragen. So können die Vorteile von GraphQL jedoch nicht richtig ausgenutzt werden, da unabhängig von den tatsächlich benötigten Daten jedes Mal alle Attribute der Rechnungsadresse angefordert werden.

Diese Problematik spiegelt sich ebenfalls an anderen Stellen in der Architektur wider. So gibt es beispielsweise für Produkte, Kundendaten, Adressen, Preise, Warenkörbe und Bestellungen jeweils einen Microservice, in dem Logik implementiert wurde, um spezielle Prozesse des Shops abzubilden und jeweils einen Adapter in Richtung commercetools, um die Daten von commercetools anzufragen und diese in ein plattforminternes Format umzuwandeln. Auf diese Weise entstehen Service-Strecken, um die Daten zu verarbeiten und für ein Frontend bereitstellen.

Mit steigender Erfahrung wurde GraphQL und damit auch das Apollo-Framework vermehrt eingesetzt. Jeder einzelne Microservice wurde als GraphQL-Server implementiert. So wird aus jeder dieser Service-Strecken ein eigener Graph mit jeweils eigener GraphQL-API und zugehörigem Schema. Um diese Graphen für ein Frontend nach außen zu vereinen, wurde ein Apollo-Gateway genutzt. Hier werden alle Schemas vereint. Werden nun in einer Anfrage an das Gateway verschiedene Ressourcen angefragt, verteilen sich die Anfragen intern parallel auf die einzelnen Graphen, um die Daten aufzulösen und bereitzustellen. So kann das Frontend und anderen Clients genau entscheiden welche Daten benötigt werden und diese gezielt anfragen.

3.2 Probleme beim Einsatz von GraphQL

In den in Kapitel 3.1 angerissenen Szenarien sind drei entscheidende Probleme entstanden. Das grundlegende Problem liegt hier zunächst in der Anbindung der GraphQL-API von commercetools. Das nächste Problem liegt bei der GraphQL-API von commercetools selbst. Diese bietet keinen Support für das Apollo-Framework. Somit sind ohne Weiteres keine Features des Apollo-Frameworks nutzbar. Das letzte für diese Arbeit relevante Problem liegt in der Natur von GraphQL und sich gegenseitig referenzierenden Datentypen. Diese Probleme sollen in den folgenden Kapiteln genauer erläutert und diskutiert werden.

3.2.1 Anbindung der GraphQL-API von commercetools

Bei der Anbindung der commercetools-Schnittstelle werden aktuell in der dotSource GmbH statische Anfragen genutzt. In Kapitel 3.2.1 wurde in Abbildung 32 ein Ausschnitt eines Microservices im produktiven Einsatz gezeigt. Es besteht potenziell das Problem des *over-fetching*, da mehr Daten von commercetools angefragt werden, als tatsächlich im Microservice benötigt werden.

Die Ursache dieses Problems liegt zum Teil an dem commercetools-Framework, welches innerhalb der JavaScript-Anwendungen genutzt wird, um Anfragen in Richtung commercetools zu senden. Es bietet keine weitere Funktionalität, um deren GraphQL-Schnittstelle anzubinden, wird aber benötigt, um die Authentifizierung an der API von commercetools vorzunehmen. Da jede Anfrage an commercetools einen Token zur Authentifizierung enthalten muss. Um eine Anfrage an die GraphQL-API zu machen, muss nur angegeben werden, dass die Anfrage an den GraphQL-Endpunkt */graphql* gehen soll. Dabei wird die *Query* als Zeichenkette im *Body* der Anfrage übergeben.

Da es in den Kundenprojekten bisher keine Möglichkeit gab, die in einen Microservice eingehenden Anfragen direkt an commercetools weiterzuleiten wurde die Funktion aus Abbildung 32 genutzt, um statische Zeichenketten zu versenden. Folgendes Schema soll das Problem verdeutlichen.

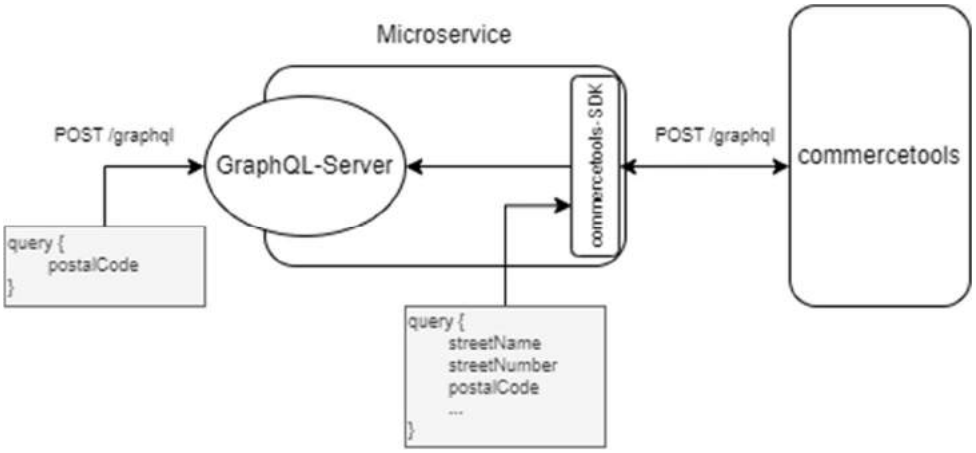


Abbildung 33: Ist-Stand - Überblick

Der in dem Microservice befindliche GraphQL-Server nimmt eine kleine *Query* entgehen, muss aber für die Anfrage an commercetools eine größere statische *Query* versenden. Da nicht bekannt welche Attribute des Objektes benötigt werden, muss es zunächst komplett angefragt werden. Es muss also eine Möglichkeit gefunden werden, die in den GraphQL-Server eingehenden Anfragen direkt an das commercetools-SDK zu übergeben, um sie direkt an commercetools weiterleiten zu können. Folgendes Schema zeigt die zu implementierende Lösungsmöglichkeit.

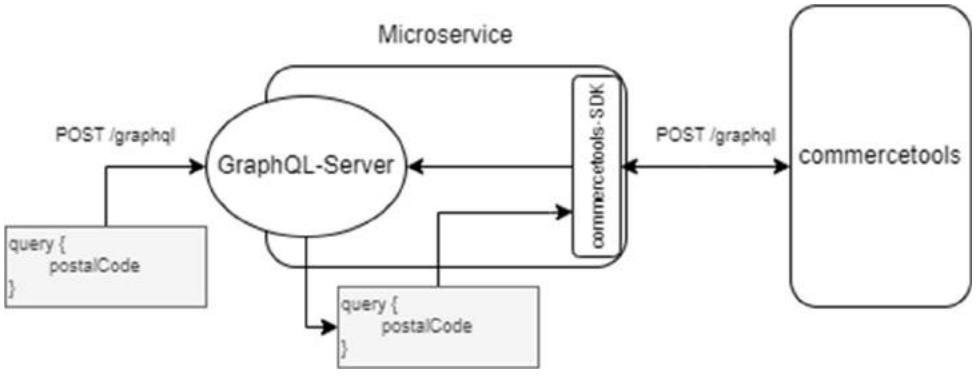


Abbildung 34: Soll-Stand - Überblick

3.2.2 Apollo Federation und commercetools

Die GraphQL-API von commercetools unterstützt keine Direktiven der Apollo Federation und entspricht auch sonst nicht der Spezifikation von Apollo, um deren Federation Feature nutzen zu können. Commercetools ermöglicht es zwar Referenzen in einem Datentyp direkt aufzulösen, jedoch ist es nicht möglich das Schema mit externen Daten aus anderen Datenquellen anzureichern. Diese müssen aktuell aufwändig in den einzelnen Services dereferenziert und zusammengeführt werden, um solche komplexeren Anfragen von außen zu ermöglichen. In Business-to-Business-Shopsystemen ist es üblich, dass die Kunden eines des Shops im Namen ihrer Firma Einkäufe tätigen. Diese Kundendaten werden mit Hilfe der *customer*-Objekte in commercetools abgebildet. Um die Daten der Firma dieses Mitarbeiters zu speichern gibt es aktuell im commercetools keinen passenden Datentyp. Hier ergibt sich der Use-Case, den *Customer*-Typ um ein Attribut zu erweitern und somit die Firmendaten direkt mit den Kundendaten zusammenzuführen. So können z.B. sehr leicht alle anderen Mitarbeiter der gleichen Firma gefunden werden. Abbildung 35 zeigt eine schematische Darstellung einer Architektur, wie sie mit Hilfe von Apollo Federation möglich wäre.

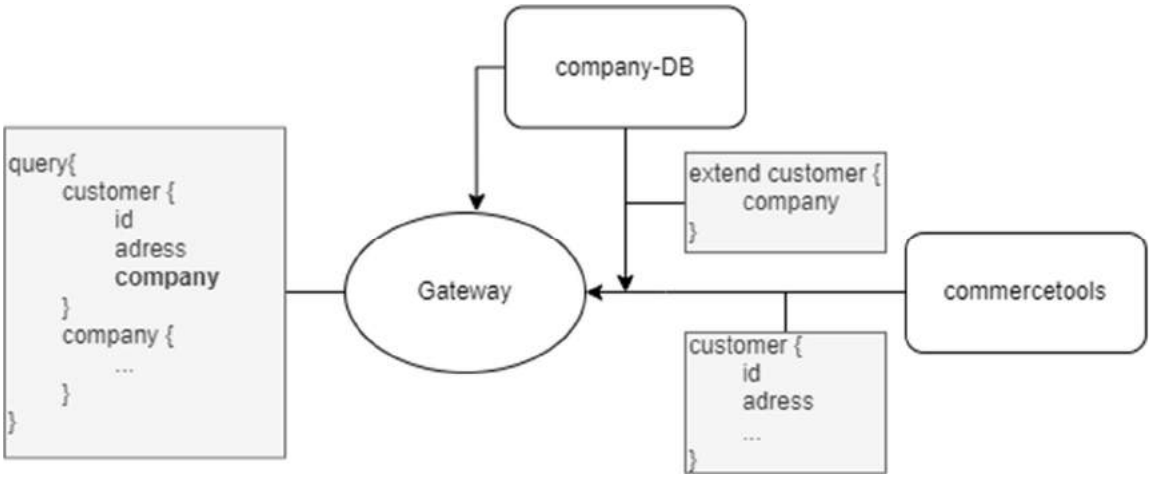


Abbildung 35: Apollo Federation - Soll-Zustand

Der *Customer*-Typ wird von commercetools mit all seinen Attributen, wie z.B. *id* und *address*, bereitgestellt. Eine externe Datenbank speichert alle Firmendaten zu den registrierten Mitarbeitern (*customer*) und stellt ein Schema für einen *Company*-Typ zur Verfügung. Der *customer*-Typ soll um ein Attribut *company* erweitert werden, so dass es in einem gemeinsamen Schema hinter einem Gateway nutzbar ist *(im Abbild fett gedruckt*). Des Weiteren soll *Company* in diesem Schema ebenfalls als allein stehender Typ Verfügbar sein. Dazu soll das commercetools-Schema dynamisch um entsprechende Direktiven erweitert werden. Eine Lösungsmöglichkeit für diese Erweiterung wird in Kapitel 4.3.2 implementiert.

3.2.3 **Das N+1-Problem in verteilten Graphen**

Dieses Problem liegt in der Natur von GraphQL selbst, wenn ein Datentyp einen anderen beispielsweise über eine ID als Attribut referenziert und dieser Datentyp an einer anderen Schnittstelle angefragt werden muss, da die ursprüngliche Schnittstelle nicht genügend Informationen bereitstellt. [41] Auf die Weise kann ein Schema mit Hilfe von Apollo Federation erweitert werden. So kann wie beispielsweise in Kapitel 3.2.2 erklärt der Customer-Typ um ein Attribut *company* erweitert werden, welches vom gleichnamigen Typ *Company* ist und vollständig gemeinsam mit einem *customer* abgefragt werden kann. Sollte nun eine Liste von 10 *customer*-Objekten an dem in Abbildung 35 abgebildeten Gateway angefragt werden, kann das Gateway diese 10 *customer* direkt innerhalb einer Anfrage von commercetools angefordert werden. Diese Objekte enthalten zu diesem Zeitpunkt jedoch noch keine Informationen zur *company*. Im nächsten Schritt muss das Gateway für jedes einzelne *customer*-Objekt aus der Liste dessen zugeordnetes *company*-Objekt aus der *company*-Datenbank abfragen. Dazu werden 10 Anfragen an diese Datenbank benötigt. Insgesamt sind also für eine Liste von 10 Kundendaten 11 Anfragen (also N+1 Anfragen) notwendig. Für dieses Problem gibt es bereits einige Lösungen, um alle nötigen Anfragen in einer zusammenzufassen. Jedoch ist es bisher nicht möglich gewesen diese in eine solche Microservicearchitektur zu integrieren. Des Weiteren soll untersucht werden, ob das sogenannte *Batching* also das Zusammenfassen der nötigen Anfragen einen Performancegewinn bringt. Da die aktuellen Lösungen darauf beruhen einige CPU-Zyklen zu warten und alle anfallenden Anfragen abzufangen um diese anschließend in eine einzige Anfrage zu bündeln und zu versenden. Es besteht also die Möglichkeit hier mehr Zeit durch das Warten zu verlieren, als durch die gesparten Anfragen gewonnen werden kann. In Kapitel 4.4 wird eine mögliche Lösung implementiert.

[41] Vgl. ebenda, S. 339.

4 Ein Lösungsansatz

Die in Kapitel 3.2 aufgezeigten Probleme sollen nun in Form der folgende Forschungsfragen untersucht werden:

- *Kann die externe GraphQL-API von commercetools an einen eigenen GraphQL-Service angebunden werden, ohne dabei vorprogrammierte statische Queries in Richtung commercetools zu verwenden?*
- *Kann die externe GraphQL-API von commercetools, welche nicht der Spezifikation von Apollo entspricht, um das Feature Federation 2.0 zu nutzen, vollautomatisiert in eine eigene Microservicearchitektur integriert werden, so dass die Typen der externen API von commercetools via Federation an einem eigenen Apollo-Gateway zur Verfügung stehen?*
- *Ist es weiterhin möglich die Performance komplexer Anfragen zu erhöhen, in dem das sogenannte „N+1"-Problem in einer möglichen Lösung mittels Batching umgangen wird?*

Um diese Fragen zu klären, folgt in den nächsten Kapiteln eine mehrstufige Lösung. Zunächst muss die GraphQL-API von commercetools angebunden werden, um das Schema der Schnittstelle zu erhalten. Dieses Schema muss anschließend so angepasst werden, dass es der Spezifikation für Federation entspricht. Zuletzt müssen die *Resolver* alle Anfragen an commercetools weiterleiten. Dabei müssen sie jedoch ebenfalls den Spezifikationen für Federation entsprechen und das „N+1"-Problem zu beheben. Am Ende soll also ein Service entstehen, welcher als Adapter zwischen den Microservices und commercetools alle benötigten Typen via Federation bereitstellt, das „N+1"-Problem umgeht, ohne dabei statisch formulierte Anfragen zu verwenden. Dieser Service soll im Folgenden apollo-commercetools-adapter heißen.

4.1 Aufbau der Testumgebung

Um diesen Service zu entwickeln und zu testen wurde zunächst eine Testumgebung eingerichtet. Diese besteht aus einem Kubernetes-Cluster in der Microsoft Azure Cloud. Die darin enthaltenen Services wurden in NodeJs implementiert. Diese sollen wichtige Ausschnitte einer Microservicearchitektur abbilden. Dazu wurde zuerst die nicht relationale Datenbank MongoDB im Kubernetes-Cluster eingerichtet. Die dort enthaltenen Objekte sollen die Datentypen von commercetools später erweitern um die Funktionalität der Apollo Federation in Verbindung mit dem zu entwickelnden apollo-commercetools-adapter zu testen. Des Weiteren wurde ein GraphQL-Service entwickelt, um die Daten aus der MongoDB als GraphQL Schema innerhalb des Clusters bereitzustellen. In einem Apollo-Gateway sollen nun das Schema des GraphQL-Service der MongoDB und das Schema des zu entwickelnden apollo-commercetools-adapters zusammengeführt werden. Dabei soll das Apollo-Gateway die erweiterten Datentypen von commercetools auflösen und bereitstellen. Folgendes Abbild soll die Architektur schematisch darstellen.

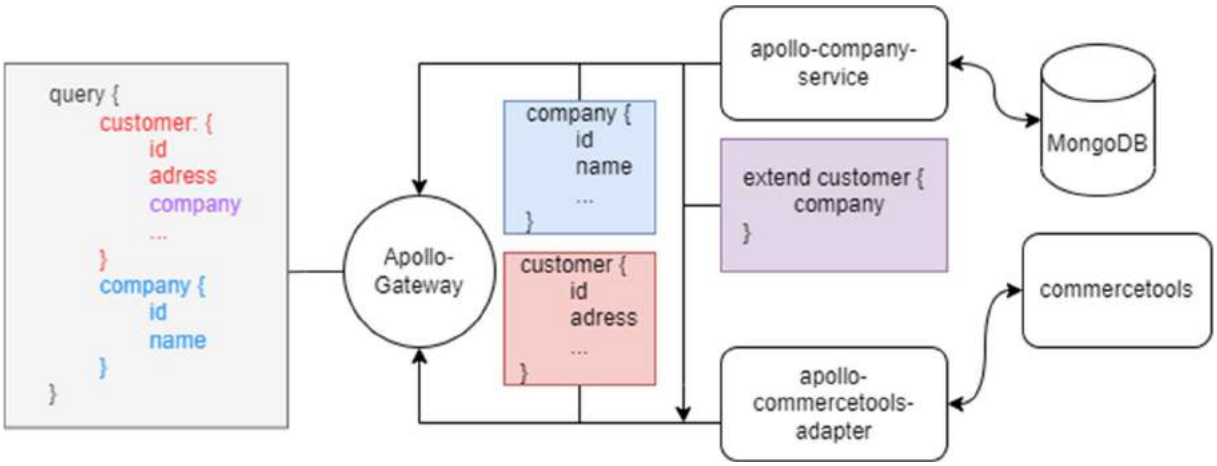


Abbildung 36: Testumgebung - Überblick

Dabei ist der in Abbildung 36 gezeigte apollo-commercetools-adapter als Hauptteil der Lösung implementieren. Dieser soll das commercetools-Schma und insbesondere den Typ *Customer* im Rahmen von Apollo Federation bereitstellen. So kann dieser Typ vom apollo-company-service um ein Attribut *company* erweitert werden. Ein Apollo-Gateway soll das zusammengesetzte Schema anschließend bereitstellen. Die Implementation des apollo-company-service soll, bis auf die Erweiterung um die *dataloader*-Bibliothek nicht weiter betrachtet werden. Sie folgt einfachen und standardmäßigen Ansätzen zur Implementierung eines Apollo-Servers in Kombination mit der Bibliothek *mongoose,* um eine MongoDB anzubinden.

Aus welchen Teilen das Schema zusammengesetzt werden soll, ist dem farblichen Schema der Abbildung 36 zu entnehmen. Wie bereits erwähnt, ist der apollo-company-service, um einen Dataloader zu erweitern. Somit kann evaluiert werden ob das „N+1“-Problem innerhalb dieser Architektur auf diese Weise gelöst werden kann. Zuletzt ist ein Apollo-Gateway aufzusetzen, um das zusammengesetzte Schema zu testen.

Auf diese Weise kann ein reales Szenario eines Kundenprojektes simuliert werden, in dem Daten durch andere externe Datenquellen in ein bestehendes GraphQL Schema basierend auf der GraphQL-API von commercetools integriert werden.

4.2 Anbindung der GraphQL-API von commercetools

Die Lösung des ersten Schrittes zum apollo-commercetools-adapter liegt in der Anbindung der GraphQL-API von commercetools selbst. Hierzu ist zuallererst eine Authentifizierung notwendig. Jede API von commercetools ist über einen Token abgesichert.[42] Dieser muss in jeder Anfrage an commercetools mitgesendet werden. Ein Token wird gegen ein *client-secret* und eine *client-id* an der Authentifizierungs-API von commercetools ausgestellt und muss regelmäßig erneut werden.[43] Um das Token zu verwalten, zu erneuern und jeder Anfrage hinzuzufügen, stellt commercetools ein Software Development Kit unter anderem auch für NodeJs zur Verfügung.[44] Dafür sind einige Funktionen aus verschiedenen Bibliotheken notwendig. Folgender Ausschnitt zeigt die für den apollo-commercetools-adapter importierten Funktionen.

```
1  import { createClient } from '@commercetools/sdk-client';
2  import { createAuthMiddlewareForClientCredentialsFlow } from '@commercetools/sdk-middleware-auth';
3  import { createHttpMiddleware } from '@commercetools/sdk-middleware-http';
4  import 'isomorphic-fetch';
```

Abbildung 37: apollo-commercetools-adapter – commercetools SDK

Diese Bibliotheken können über den *node package manager* mit den folgenden Befehlen installiert werden.

```
npm install @commercetools/sdk-client
npm install @commercetools/sdk-middleware-auth
npm install @commercetools/sdk-middleware-http
npm install isomorphic-fetch
```

[42] [com 22a].
[43] Vgl. ebenda.
[44] [com 22b].

Folgende Abbildung zeigt einen Ausschnitt einer Datei des apollo-commercetools-adapters, die ein Objekt erzeugt, welches alle notwendigen Konfigurationen enthält, um mit commercetools zu kommunizieren.

```
8   function createCommercetoolsClient() {
9       const {
10          authHost, httpApiHost, projectKey, scope, clientId, clientSecret,
11      } = provideConfiguration();
12      return createClient({
13          middlewares: [
14              createAuthMiddlewareForClientCredentialsFlow({
15                  host: authHost,
16                  projectKey,
17                  credentials: {
18                      clientId,
19                      clientSecret,
20                  },
21                  scope,
22              }),
23              createHttpMiddleware({
24                  host: httpApiHost,
25              }),
26          ],
27      });
28  }
```

Abbildung 38: apollo-commercetools-adapter – commercetools I

Die *provideConfiguration*-Funktion in Zeile 11 liest alle Werte, wie z.B. das *client-secret* und die *client-id* aus Umgebungsvariablen aus. Anschließend werden sie der *createClient*-Funktion in einem Objekt zur Konfiguration übergeben. Diese Funktion wird vom commercetools-SDK bereitgestellt. Der Rückgabewert dieser Funktion ist ein *client*-Objekt, welches alle Parameter zur Authentifizierung enthält und diese an die Anfragen in Richtung commercetools anfügt. Um nicht bei jeder Anfrage ein neues *client*-Objekt zu erstellen, wird es zwischengespeichert und bei jeder Anfrage das Objekt aus dem Speicher genutzt. Folgende Funktion stellt dieses zwischengespeicherte Objekt bereit.

```
function getCommercetoolsClient() {
    if (!cachedClient) {
        cachedClient = createCommercetoolsClient();
    }
    return cachedClient;
}
```

Abbildung 39: apollo-commercetools-adapter - Client

Dazu wird die bereits gezeigt Funktion *createCommercetoolsClient* verwendet und in der Variable *cachedClient* gespeichert. Diese Funktion gibt ausschließlich den Wert aus *cachedClient* zurück. Sollte diese Variable nicht gesetzt sein, wird ein *client* erzeugt und darin gespeichert. So kann in anderen Dateien immer auf den gleichen *client* zugegriffen werden, um Anfragen an commercetools zu stellen.

Da der apollo-commercetools-adapter als Schnittstelle für andere Microservices in Richtung commercetools dienen soll, müssen zwei grundlegende Probleme gelöst werden. Zum einen muss das Schema der GraphQL-API durch den apollo-commercetools-adapter innerhalb der Microservicearchitektur bereitgestellt werden und zum anderen müssen alle Anfragen von diesem Service an commercetools weitergeleitet werden, ohne dabei statische Anfragen zu nutzen. In Kapitel 3.2.1 wurde erwähnt, wie bisher sowohl das Schema von commercetools manuell und statisch in einer Datei gepflegt wurde, um es dem GraphQL-Server bereitzustellen, als auch die Anfragen in Form einer statischen Zeichenkette implementiert wurden, um sie an commercetools zu senden. Das sorgt für höheren Implementierungs- und Wartungsaufwand und führt zu *over-fetching* an der commercetools-API.

Um diese Probleme zu lösen können zwei Funktion der Bibliothek *graphql-tools* genutzt werden. Diese Bibliothek kann ebenfalls über den *node package manager* installiert werden, um anschließend die Funktionen *wrapSchema* und *introspectSchema* zu importieren. Diese Funktionen helfen dabei nicht veränderliche externe Schemas in einen eigenen GraphQL-Server einzubinden. Dazu wurde das Beispiel der Dokumentation dieser Funktionen von *graphql-tools* so angepasst, dass es das zuvor erzeugte *client*-Objekt für commercetools nutzt, um das Schema mit all seinen Typen mittels *introspection* abzufragen. Daraus ergibt sich folgende Implementierung.

```
22  async function getCommercetoolsSchema () {
23    const schema = wrapSchema({
24      schema: await introspectSchema(executor),
25      executor
26    })
27    return schema
28  }
```

Abbildung 40: apollo-commercetools-adapter – commercetools II

Die hier definierte Funktion *getCommercetoolsSchema* soll das GraphQL Schema von commercetools abfragen und es zurückgeben. Dazu wird die Funktion *wrapSchema*-Funktion aufgerufen und deren Rückgabewert direkt zurückgegeben. Diese Funktion bekommt als Parameter ein Objekt übergeben, um Konfigurationen vorzunehmen. Dieses Objekt enthält das Schema, welches hier direkt mittels *introspectSchema* von commercetools abgefragt wird und einen *executor*. Hierbei handelt es sich um eine selbstdefinierte Funktion, in der die Anfragen in Richtung einer GraphQL-API implementiert werden. Dieser *executor* wird ebenfalls der *intosprectSchema*-Funktion übergeben, da die Abfrage des Schemas auf die gleiche Art und Weise bei commercetools durchgeführt wird, wie später alle anderen produktiven GraphQL-Anfragen. Es wäre auch möglich das Schema direkt mit der *introspectSchema*-Funktion abzurufen und es danach nicht an die *wrapSchema*-Funktion zu übergeben. Die *wrapSchema*-Funktion dient hier nur als Zwischenschritt und erfüllt keinen unmittelbaren Zweck. Sie bietet jedoch die Möglichkeit, später im Produktivbetrieb eines Kundenprojektes das abgefragte Schema von commercetools mittels eines dritten Parameters zu transformieren, also zu verändern. Dazu muss lediglich ein Parameter namens *tranformers* neben den anderen Parametern *schema* und *executor* übergeben werden. Dieser enthält dann eine Liste von Funktionen, welche das Schema verändern, indem sie Typen Filtern oder Umbenennen. So kann ein nicht veränderliches externes Schema erst abgefragt und später genauer den Bedürfnissen in einem Projekt angepasst werden. Die *executor*-Funktion ist also wie folgt definiert.

```
5  async function executor ({ document, variables }) {
6    const query = print(document)
7
8    const requestConfig = {
9      uri: `/${provideConfiguration().projectKey}/graphql`,
10     method: 'POST',
11     body: { query, variables }
12   };
13
14   const result = await getCommercetoolsClient().execute(requestConfig)
15
16   if(result.statusCode =/= 200)
17     throw new Error("An unexpected error occurred")
18
19   return result.body
20 }
```

Abbildung 41: apollo-commercetools-adapter - Executorfunktion

Die Funktion bekommt als Parameter ein Objekt mit zwei Attributen übergeben. Bei *document* handelt es sich um ein komplexeres Objekt, welches alle Informationen für eine Abfrage in der *query language* von GraphQL enthält. Sollten in dieser Anfragen Variablen verwendet werden, sind diese in *Variables* gespeichert. Mit Hilfe der *print*-Funktion aus der Bibliothek *graphql*, kann aus dem komplexen Objekt *document* in Zeile 6 die Anfrage mit all ihren Informationen eine einfache Zeichenkette erzeugt werden, welche die GraphQL *Query* enthält. In Zeile 14 wird die bereits erklärte *getCommercetoolsClient*-Funktion aufgerufen, welche ein *client*-Objekt für Anfragen an commercetools zurückgibt. Dieses Objekt verfügt über eine Funktion *execute*. Diese Funktion bekommt als Parameter alle Informationen zu einer Anfrage um diese in Richtung commercetools zu senden. Das übergebene Objekt *requestConfig* wird in Zeile 8 erzeugt und besteht aus der *uri*, welche auf die GraphQL-API von commercetools zeigt, aus der *method*, welche bei GraphQL immer *POST* ist und dem Inhalt der Anfrage, der *Query* als Zeichenkette und ihren Variablen. Mit Hilfe dieser Funktion, kann zum einen die *introspectSchema*-Funktion eine *Query* an commercetools senden, um das Schema der GraphQL-API abzufragen und zum anderen können alle in den apollo-commercetools-adapter eingehenden *Queries* direkt an diese Funktion übergeben, um sie ebenfalls an commercetools weiterzuleiten. Auf diese Art und Weise müssen die Anfragen an commercetools nicht als statische Zeichenkette implementiert werden, da diese hier von außen kommend weitergeleitet werden können. Im nächsten Kapitel wird außerdem gezeigt, wie das hier abgefragte Schema der GraphQL-API von commercetools dynamisch an den GraphQL-Server von Apollo übergeben werden kann. So muss dieses ebenfalls nicht aufwändig manuell gepflegt werden.

4.3 Apollo Federation und commercetools

4.3.1 Implementierung im apollo-commercetools-adapter

Aufbauend auf dem letzten Kapitel soll nun das von commercetools abgefragte Schema im apollo-commercetools-adapter so konvertiert werden, dass es mittels Apollo Federation in die eigene Architektur eingebunden werden kann. Die Dokumentation zu Apollo Federation benennt einige Voraussetzungen, die ein Service erfüllen muss, um ihn zu einem Subgraphen zu machen und in einen Supergraphen zu integrieren. Bei Recherchen fällt auf, dass dieses Problem in der Welt von GraphQL sehr wenig diskutiert wird und es aufgrund dessen keine populären Lösungsansätze gibt. Neben der Möglichkeit selbst eine Bibliothek zur Konvertierung von GraphQL Schemas zu implementieren, ergab sich noch eine Möglichkeit eine Bibliothek namens *graphql-transform-federation* zu nutzen. Diese kaum bekannte Bibliothek wurde von einem Entwickler im Jahr 2020 entwickelt und seitdem nicht weiter gewartet und bekommt auf GitHub auch keine größere Aufmerksamkeit.[45] Da sich GraphQL mit all seinen Bibliotheken wie Apollo seitdem sehr stark weiterentwickelt hat, ist diese Bibliothek ohne Weiteres kaum nutzbar. Diese Bibliothek ist in TypeScript geschrieben, nutzt veraltete Typen und alte Versionen der GraphQL-Bibliotheken.[46] Also wurde diese Bibliothek zunächst heruntergeladen und in ein privates Repository in Gitlab abgelegt, um einige Anpassungen vorzunehmen und Fehler zu beheben. Es wurden die dort verwendeten Bibliotheken aktualisiert und die damit einhergehenden Änderungen am Programmcode, wie z.B. die Einführung neuer Typen und Objekte, vorgenommen.

[45] [Oos 22].
[46] Vgl. ebenda.

Anschließend kann die angepasste Bibliothek ebenfalls über den *node package* manager aus Gitlab heraus installiert und importiert werden. Diese Bibliothek stellt eine wesentliche Funktion zur Konvertierung eines standardmäßiges GraphQL Schemas zur Verfügung. Die Folgende Abbildung zeigt, wie diese Funktion in den apollo-commercetools-adapter implementiert wurde.

```
1  import { transformSchemaFederation } from 'graphql-transform-federation';
2  import { delegateToSchema } from '@graphql-tools/delegate';
3
4  const options = {
5      Customer: {
6        extend: false,
7        keyFields: ['id'],
8        fields: { },
9        __resolveReference({ id }, context, info) {
10         return delegateToSchema({
11           schema: info.schema,
12           operation: 'query',
13           fieldName: 'customer',
14           args: {
15             id,
16           },
17           context,
18           info,
19         });
20       },
21     }
22 };
23
24 export default function transformSchema(schema){
25   return transformSchemaFederation(schema, options)
26 }
```

Abbildung 42: apollo-commercetools-adapter - Schemakonvertierung

Diese Abbildung zeigt eine vollständige Datei, welche am Ende eine Funktion zur Konvertierung eines GraphQL Schema dient. Dazu wird in Zeile 24 eine Funktion *transformSchema* exportiert. Diese nimmt als Parameter ein GraphQL Schema als Zeichenkette entgegen. Dabei ruft sie direkt, die in Zeile 1 importierte Funktion *transformFederationSchema* aus der bearbeiteten Bibliothek *graphql-tranform-federation* auf. Sie reicht außerdem den zuvor erhaltenen Parameter *Schema* weiter, fügt ein zusätzlichen Parameter *options* hinzu und gibt dabei direkt deren Rückgabewert zurück. Hier wird also die Funktion aus der Bibliothek und der Parameter *options* nach außen gekapselt, so dass nur noch ein Schema übergeben werden muss. Der Parameter *options* legt fest, wie das Schema konvertiert werden soll. Dazu wird in Zeile 5 ein Objekt definiert, dass zunächst ein Attribut enthält. Bei dem Namen dieses Attributes *Customer* handelt es sich ebenfalls um einen

Namen eines Typs des GraphQL Schema, welches konvertiert werden soll. In diesem Attribut ist definiert, wie der Typ angepasst werden muss, um ihn in einen Graphen mit Apollo Federation zu integrieren. Da dieser Typ nicht in Rahmen des Schemas des apollo-commercetools-adapters erweitert soll, sondern in dem GraphQL-Service für die MongoDB, ist der Parameter *extend* auf *false* gestellt. Weiterhin muss in der Liste *keyFields* angegeben werden, welche Attribute des Schemas, das Objekt, welches das Schema beschreibt, eindeutig identifizieren. Da jedes Objekt in commercetools über eine ID verfügt, wird diese hier zunächst angegeben. Commercetools bietet weitere Möglichkeiten Objekte vom Typ *Customer* zu identifizieren, auf die innerhalb dieses Tests jedoch verzichtet wird. Zuletzt wird dem Objekt in Zeile 9 ein zusätzlicher *Resolver*, wie sie in Kapitel 2.2 erklärt wurden, hinzugefügt. Diese *Resolver*-Funktion erhält hier jedoch den Namen *__resolveReference*, da die Spezifikation von Apollo Federation vorgibt, dass beim Dereferenzieren fremder Typen durch das Apollo-Gateway bei dem für den Typen zuständigen GraphQL-Service, ein *Resolver* namens *__resolveReference* aufgerufen wird. Innerhalb dieser Funktion kommt eine weitere Funktion aus der Bibliothek *graphql-tools* mit dem Namen *delegateToSchema* zum Einsatz. Diese Funktion ermöglicht es GraphQL-*Queries* an eine andere GraphQL-API weiterzuleiten. Dieser Funktion werden beim Aufruf ebenfalls einige Optionen in Form eines Objektes übergeben. Sollte also im apollo-commercetools-adapter eine Anfrage von einem Apollo-Gateway eingehen, um ein Objekt vom Typ *Customer* mittels einer *ID* aufzulösen, ist hier definiert, dass diese Anfrage an die GraphQL-API von commercetools weitergeleitet wird. Dazu soll das Feld *customer* in dem Schema von commercetools genutzt werden. In dem *args*-Parameter wird die *Id* als Argument an das *customer*-Feld des commercetools Schemas übergeben. Auf diese Weise kann commercetools das Objekt an den apollo-commercetools-adapter zurückgeben. Dieser kann es anschließend an das Apollo-Gateway weiterleiten, um diese Typen dort zusammenzuführen.

Nachdem nun die Schemadefinition konvertiert wurde, kann ein Apollo-Server im apollo-commercetools-adapter implementiert werden. Jetzt setzen sich alle implementierten Funktionen aus diesem und dem letzten Kapitel zusammen und der Apollo-Server wurde wie folgt implementiert.

```
1  import { ApolloServer } from 'apollo-server';
2  import getSchema from './schema/commercetools.js';
3  import transformSchema from './schema/transform.js';
4
5  const schemaWithoutFederation = await getSchema()
6  const federatedSchema = transformSchema(schemaWithoutFederation)
7
8  const server = new ApolloServer({
9    schema: federatedSchema
10 })
11
12 server.listen({port: 8080})
```

Abbildung 43: apollo-commercetools-adapter - Apollo-Server

Die Abbildung zeigt die Datei, welche beim Start des apollo-commercetools-adapters aufgerufen wird. In Zeile 2 wird also die in Kapitel 4.2 implementierte Funktion *getSchema* zur Abfrage des Schemas von commercetools importiert. Anschließend wird in Zeile 3 die zuvor erläuterte Funktion *transformSchema* zur Konvertierung eines Schemas importiert. Beim Start des apollo-commercetools-adapters wird also in Zeile 5 das GraphQL Schema von commercetools abgefragt und in *schemaWithoutFederation* gespeichert, Anschließend wird in Zeile 6 dieses Schema in ein Schema konvertiert, welches der Spezifikation von Apollo Federation entspricht. Hier ist unbedingt zu erwähnen, dass commercetools eine Vielzahl verschiedener Typen, wie z.B. *order, cart* oder *shoppingList* in diesem Schema definiert. Alle Typen von commercetools sind nun ebenfalls in dem apollo-commercetools-adapter verfügbar und können direkt über diesen angefragt werden. Jedoch wurde, um die Komplexität zu verringern, nur der Typ *customer* im apollo-commercetools-adapter so angepasst, dass er mit Apollo Federation kompatibel ist.

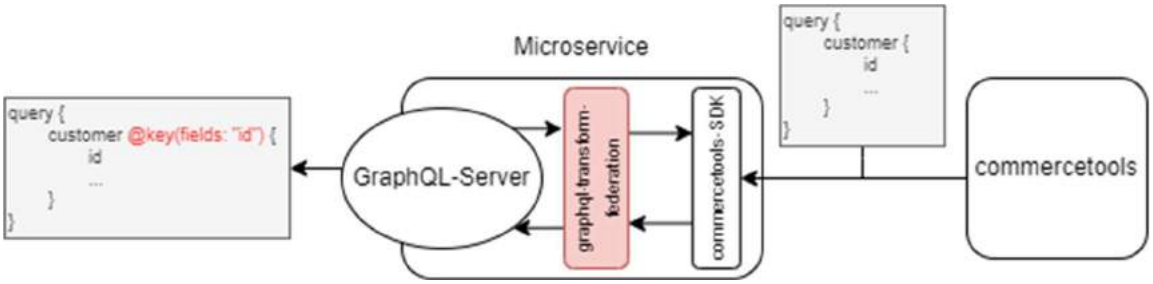


Abbildung 44: apollo-commercetools-adapter- Schema

In Abbildung 44 ist schematisch zu sehen, wie das ursprüngliche Schema von commercetools vom apollo-commercetools-adapter abgerufen wird. Dieser Microservice wandelt das Schema mit der in Rot gekennzeichnete Bibliothek um und fügt unter anderem die ebenfalls in Rot gekennzeichnete Direktive *key* hinzu. So ist das Schema für Apollo Federation bereit.

4.3.2 Erweiterung des Typs *Customer*

Da in Kapitel 4.3.1 der *customer*-Typ im Schema des apollo-commercetools-adapters um eine *__resolveReference*-Funktion und die *key*-Direktive erweitert wurden, ist dieser Typ zu einer Entität geworden, welche im Rahmen von Apollo Federation nutzbar ist. So kann nun im apollo-company-service der *customer*-Typ wie folgend erweitert werden.

```
17  extend type Customer @key(fields: "id"){
18    id: String! @federation__external
19    company: Company
20  }
```

Abbildung 45: apollo-company-service - *customer*-Typ

Wie bereits bekannt handelt es sich bei dem Typ *customer* um einen von commercetools definierten Typen. Diese Abbildung zeigt einen Ausschnitt einer Schemadefinition eines externen Typs im apollo-company-service, welcher hier mit dem Schlüsselwort *extend* erweitert wurde. Dadurch ist dem Apollo-Gateway später bekannt, dass es sich um eine Erweiterung eines anderen Typs handelt und die vollständige Definition dieses Typs in einem anderen Service, in diesem Fall dem apollo-commercetools-adapter, zu finden ist. Die Direktive *federation_external* beschreibt, dass auch die ID von außen bereitgestellt wird. Zu jeder *Company* ist in der MongoDB die ID eines *customers* als Sekundärschlüssel gespeichert. Hier wird also nur das Feld *company* mit dem Typ *Company* dem bereits vorhandenem *Customer*-Typ des apollo-commercetools-adapters hinzugefügt. Zu dem neuen Feld muss anschließend noch ein passender *Resolver* im apollo-company-service implementiert werden.

Im nächsten Ausschnitt ist ein *Resolver* für das Feld *company* im Typ *customer* definiert.

```
45      Customer:{
46        company: async (customer) =>{
47          return Company.findOne({employee: customer.id})
48        }
49      }
```

Abbildung 46: apollo-company-service - Resolver

Da vom *customer*-Typ die Id bekannt ist, kann diese in der Anfrage für die MongoDB verwendet werden. Dazu wird die *findOne*-Funktion der Bibliothek *mongoose* genutzt, um ein *company*-Objekt zu finden, bei dem die hinterlegte Id für *employee* mit der Id des *customer*-Objektes übereinstimmt.

4.3.3 Einrichten des Apollo-Gateways

Da nun der apollo-commercetools-adapter den Voraussetzungen für Apollo Federation entspricht und im apollo-company-service ein Typ des Schemas von commercetools erweitert wurde, können in einem Apollo-Gateway beide Schemas zusammengeführt werden. Das Schema des apollo-commercetools-adapter stellt für sich genommen, nur Typen der GraphQL-API von commercetools bereit. Hier sind also die Felder *company* und *companies* nicht vorhanden, da diese ausschließlich im apollo-company-service definiert sind. Wird die GraphQL-API das apollo-commercetools-adapters also für sich allein betrachtet, sind all diese Felder nicht vorhanden. Erst durch die Zusammenführung mit dem apollo-company-service im Apollo-Gateway, werden die eben genannten Felder sichtbar und nutzbar. Die Implementierung eines solchen Gateways erstreckt sich nur über wenige Zeilen Programmcode. Folgende Abbildung zeigt die Implementierung des Apollo-Gateways mit dem zusammengefassten Schema.

```
1  const { ApolloServer } = require('apollo-server');
2  const { ApolloGateway  } = require('@apollo/gateway');
3  const { readFileSync } = require('fs');
4
5  const supergraphSdl = readFileSync('./supergraph.graphql',  'utf-8').toString();
6
7  const gateway = new ApolloGateway({supergraphSdl});
8
9  const server = new ApolloServer({
10   gateway,
11   subscriptions: false
12 });
13
14 server.listen({port: 8080}).catch(err => {console.error(err)});
```

Abbildung 47: apollo-gateway - Instanziierung

Nach einigen importierten Funktionen von Apollo wird in Zeile 5 eine Datei, welche die Definition der zusammengeführten Schemas des apollo-company-services und des apollo-commercetools-adapters enthält, vom Dateisystem eingelesen und in der Variable *supergraphSdl* gespeichert. Auch hier wird ebenfalls ein Apollo-Server in Zeile 9 erzeugt. Um aus diesem Server nun ein Gateway zu machen, wird ein Gateway-Objekt in Zeile 9 mit der Schemadefinition erzeugt in dem Server übergeben. Dieser wird anschließend in Zeile 14 gestartet. Zuletzt muss die In Zeile 5 eingelesene Datei *supergraph.graphql* erzeugt werden, bevor das Apollo-Gateway gestartet werden kann. Auch hierfür haben die Entwickler von Apollo ein Programm für die Kommandozeile namens *rover* entwickelt. Der folgende Befehl erzeugt diese Datei.

```
rover supergraph compose --config ./supergraph-config.yaml
> ./supergraph.graphql
```

Diesem Befehl wird hinter dem Parameter *config* folgende Konfiguration im yaml-Format übergeben.

```
1  federation_version: 2
2  subgraphs:
3    companies:
4      routing_url: http://apollo-company-service/graphql
5      schema:
6        subgraph_url: https://graphql.westeurope.cloudapp.azure.com/apollo/manufacturer/graphql
7    commercetools:
8      routing_url: http://apollo-commercetools-adapter
9      schema:
10       subgraph_url: https://graphql.westeurope.cloudapp.azure.com/apollo/commercetools/graphql
```

Abbildung 48: apollo-gateway - Konfiguration des Supergraphen

Diese Datei enthält die Definition der Subgraphen *commercetools* und *companies*. Für jeden dieser Subgraphen ist jeweils angegeben und welcher URL das Programm *rover* das Schema der Subgraphen abrufen kann und außerdem, wohin die GraphQL-Anfragen weitergeleitet werden sollen. Da sich das Apollo-Gateway zur Laufzeit in einem Kubernetes-Cluster befindet, handelt es sich bei den *routing_urls* um clusterinterne Adressen. Da das Apollo-Gateway die Anfragen direkt intern an die betreffenden Services weiterleiten soll. Die *subgraph_url* hingegen ist eine öffentliche Adresse, da *rover* außerhalb des Kubernetes-Clusters ausgeführt wird, um die Definitionen der Subschemas zu erhalten. Das Kubernetes-

Cluster wurde dementsprechend konfiguriert, um die Schemas öffentlich zur Verfügung zu stellen. Damit ist die in Kapitel 4.3.1 gezeigte Architektur vollständig. Es wurde der apollo-commercetools-adapter implementiert, sowie das Apollo-Gateway aufgesetzt.

4.4 Das N+1-Problem und Dataloader

Da es nun nach den vorangegangenen Schritten möglich ist über das Apollo-Gateway verschiedene *customer* abzufragen und dabei direkt die Daten der dazugehörigen *company* zu aggregieren, ergibt sich nun auch hier das „N+1"-Problem. Wenn nun am Apollo-Gateway über das Feld *customers* eine Liste von Kundendaten der Länge *n* angefragt wird, in der jedes *customer*-Objekt Daten zu der dazugehörigen *company* enthält, ist eine Anfrage an den apollo-commercetools-adapter notwendig, um die Liste aller Kunden zu erhalten und *n* Anfragen an den apollo-company-service, um für jedes *customer*-Objekt in der Liste die Daten der jeweiligen *company* zu dereferenzieren. Um zu verhindern, dass der apollo-company-service infolgedessen ebenfalls *n*-Mal eine Anfrage an die MongoDB schickt, um die Daten der Firmen abzufragen, wird in diesem Service die Bibliothek *dataloader* implementiert. Mit Hilfe dieser Bibliothek werden eingehende Anfragen über eine bestimmte Zeit gesammelt, dann zusammengefasst und können anschließend ein Einem verarbeitet werden. So ergibt sich für den apollo-company-service folgende zusätzliche Implementierung.

```
7  const Dataloader = require("dataloader")
8
9  const server = new ApolloServer({
10   schema: buildSubgraphSchema({typeDefs, resolvers}),
11   context: {
12     companyLoader: new Dataloader(async (keys) => {
13       const result = await Company.find({employee: {'$in': keys}})
14       return result
15     })
16   }
17 });
```

Abbildung 49: apollo-company-service - Implementierung des *dataloaders* I

Es ist möglich dem Apollo-Server bei der Instanziierung einen weiteren Parameter *context* zu übergeben. Dieses in Zeile 11 erzeugte Objekt steht anschließend in jeder *Resolver*-Funktion zur Verfügung. Also kann dem *context*-Objekt ein Attribut hinzugefügt werden, welches eine Instanz des *dataloaders* enthält. Bei dem Aufruf des Konstruktors wird der Dataloader-Klasse eine Callback-Funktion übergeben, in der definiert ist, wie die Daten anschließend in einer Anfrage an die MongoDB abgefragt werden sollen. Hier ist nochmal zu verdeutlichen, dass das Feld *employee* in einer *company* als Sekundärschlüssel dient, um den zugehörigen *customer* zu identifizieren. Dazu enthält das *employee*-Feld die jeweilige Id des *customer*-Objektes. Sollte der apollo-company-service nun mehrfach einzeln aufgerufen werden, um die *company* zu einem *customer* zu erhalten, werden alle IDs der *customer* aus den einzelnen Anfragen gesammelt in der Liste *keys* in Zeile 12 gespeichert. Anschließend kann eine einzige Anfrage an die MongoDB gesendet werden. In Zeile 13 werden also alle *company*-Objekte angefragt, deren ID im Feld *employee* ebenfalls in der Liste *keys* enthalten ist. Anschließend wurde der in Kapitel 4.3.2 zusätzlich eingefügte *Resolver*, wie im folgenden Ausschnitt zusehen, bearbeitet.

```
43    Customer:{
44      company: async (customer, args,{customerLoader}) => {
45        let result
46
47        if(process.env.USE_DATALOADER === 'true')
48          result = await customerLoader.load(customer.id)
49        else
50          result = await Company.findOne({employee: customer.id})
51
52        return result;
53      }
54    }
```

Abbildung 50: apollo-company-service - *dataloader*

In Zeile 50 ist nach wie vor die Logik für einzelne Abfragen an die MongoDB enthalten. Jedoch kann nun über eine Umgebungsvariable *USE_DATALOADER* gesteuert werden, ob der *dataloader* im eben erklärten *customerLoader* in Zeile 48 genutzt wird oder die einfache Variante aus Zeile 50. Sollte *USE_DATALOADER* auf *true* gesetzt werden wird die *load*-Funktion des *customerLoaders* aufgerufen. Dabei werden nach und nach alle IDs der *customer*-Objekte übergeben, um diese in der Liste *keys* zu sammeln und dann eine einzelne Anfrage an die MongoDB zu stellen. Die Auswahl der Varianten per Umgebungsvariable wurde hier nur zu Testzwecken eingefügt. So ist es später bei der Evaluierung sehr einfach möglich beide Varianten hinsichtlich der Anzahl der Anfragen an die MongoDB und der gesamten Performance zu vergleichen, da zwischen den Tests nur eine Umgebungsvariable geändert werden muss. Sonst sind keine weiteren Anpassungen am Programmcode notwendig.

5 Evaluation der Lösung

5.1 Auswertung der Anbindung an commercetools

In Kapitel 4.2 wurde ein möglicher Lösungsansatz gezeigt, wie die GraphQL-API von commercetools in einer Microservicearchitektur angebunden werden kann, ohne vorprogrammierte statische *Queries* zu verwenden. Folgende Abbildung zeigt das Apollo Studio des apollo-commercetools-adapters.

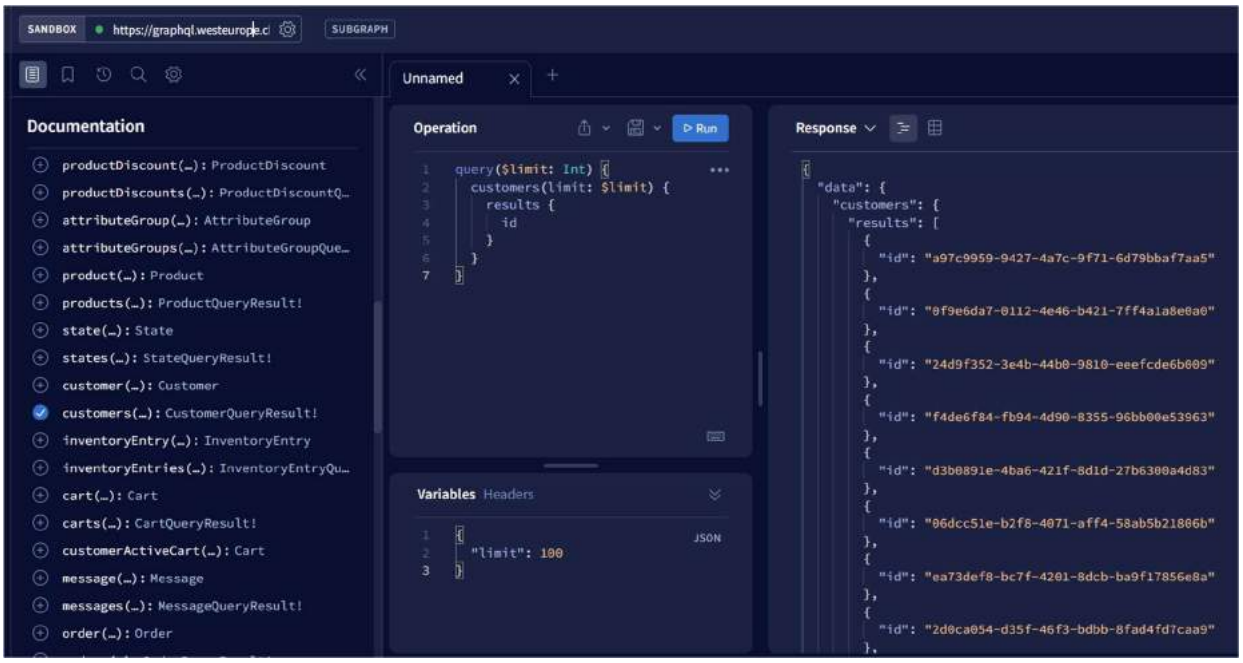


Abbildung 51: Evaluierung - Apollo Studio

Im linken Teil der Abbildung ist zu sehen, dass der apollo-commercetools-adapter alle Typen der GraphQL-API von commercetools selbst bereitstellt. Dabei wurde das Schema wie gefordert nicht manuell im Service als Datei hinterlegt. Es wurde stattdessen, wie in Kapitel 4.2 bereits erläutert, beim Start des Service einmal angefragt, um es dem Apollo-Server bei der Instanziierung bereitzustellen. Im mittleren Teil ist eine beispielhaft kleine Anfrage zusehen, welche an den apollo-commercetools-adapter gesendet und an commercetools weitergeleitet wurde. Der rechte Teil zeigt weitergeleitete Antwort von commercetools. Um zu zeigen, dass dieser Service mit dynamischen Anfragen in Richtung commercetools arbeitet, wurde folgender Test durchgeführt.

Zuerst wurde der apollo-commercetools-adapter kopiert und so angepasst, dass er eine fest definierte Zeichenkette an commercetools sendet, um eine Liste von *customer*-Objekten anzufragen. Dieser Service soll nun static-commercetools-adapter heißen. Da nicht bekannt ist, welche Attribute bei einer Anfrage an den static-commercetools-adapter erforderlich sind, muss eine Anfrage in einer Zeichenkette statisch implementiert sein und zunächst alle möglichen Attribute des *customer*-Objektes von commercetools erhalten. Die im Anhang 1 befindliche Abbildung zeigt die statische Zeichenkette des static-commercetools-adapters, welche den aktuellen Implementierungen von statischen Anfragen in Kundenprojekten sehr nahekommt**.** Es wird also jedes Mal ein sehr großes *customer*-Objekt angefragt und komplett von commercetools zurückgegeben, auch wenn die Anfrage an den static-commercetools-adapter möglicherweise nur die IDs aller Kunden anfordert und nicht das gesamte Objekt. Dieses Szenario soll der dynamischen Lösung gegenübergestellt und ausgewertet werden. Dazu wird das übertragene Datenvolumen der gesendeten Anfragen und der erhaltenen Antworten zwischen commercetools und den beiden Services verglichen. Dazu wurde ein Proxy-Server in NodeJs mit dem Express-Framework implementiert, welcher die Anfragen des apollo-commercetools-adapters und des static-commercetools-adapters an commercetools weiterleitet und dabei die Größe der Anfrage auf der Konsole ausgibt. Weiterhin wird die Antwort von commercetools ebenfalls durch diesen Proxy geleitet und deren Größe auf der Konsole ausgegeben.

Bei der Anfrage aus Abbildung 51 an den static-commercetools-adapter ergibt sich bei dem Proxy-Server folgende Konsolenausgabe.

```
static-commercetools-adapter
Bytes send:  1171  Bytes
Bytes received:  59677  Bytes
```

Abbildung 52: Datenvolumen mit statischem Schema

Es wurden also 1171 Bytes an commercetools gesendet und commercetools hat 59677 Bytes geantwortet. Bei der gleichen Anfrage an den static-commercetools-adapter, ergibt sich hingegen folgende Ausgabe.

```
apollo-commercetools-adapter
Bytes send:  454  Bytes
Bytes received:  6547  Bytes
```

Abbildung 53: Datenvolumen mit dynamischem Schema

Hier wurde mit 454 Bytes das gesendete Datenvolumen um rund 61% verringert. Auch das Datenvolumen der Antwort konnte infolgedessen deutlich optimiert werden. Mit 6547 Bytes entspricht das eine Reduktion von rund 89%. Es ist also deutlich zusehen, dass der apollo-commercetools-adapter dynamische Anfragen nutzt und somit das übertragende Datenvolumen stark optimiert. In realen Szenarien, in denen mehrere tausend Kundendaten täglich abgefragt werden, ist dieser Effekt umso größer. Da es zu erwarten ist, dass damit auch ein Performancegewinn einher geht, wurden anschließend beide Services einem Performancetest mit Hilfe des Tools *autocannon* unterzogen. Die Implementierung des Performancetests ist dem Anhang 2 zu entnehmen. Im Rahmen dieses Tests wurde die gleiche Anfrage wie in dem letzten Test genutzt (siehe Abbildung 51). Beide Services wurden für 30 Sekunden mit 5 parallelen Verbindungen mit so vielen Anfragen wie möglich unter Last gesetzt. Dabei ergab sich für den static-commercetools-adapter folgendes Ergebnis.

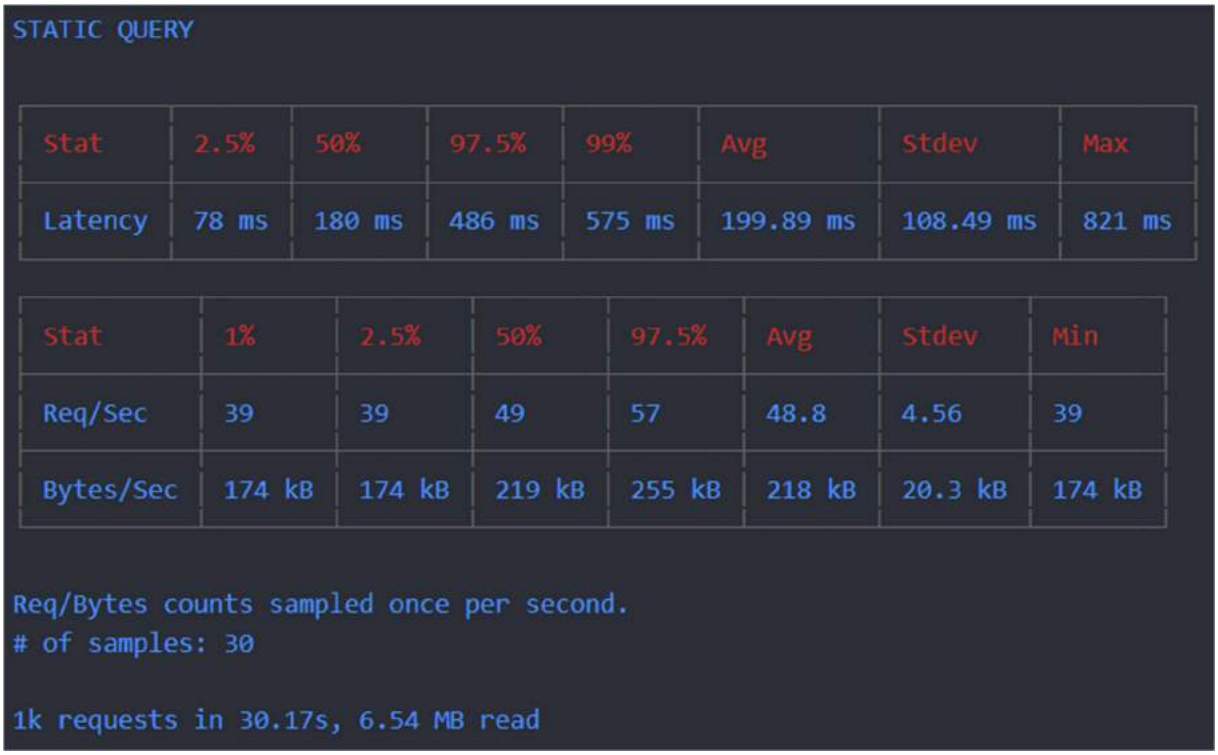


Abbildung 54: Performancetest mit statischem Schema

Der Performancetest ergab, dass der static-commercetools-adapter in der Lage war über eintausend Anfragen zu bearbeiten. Dabei lag die durchschnittliche Antwortzeit bei ca. 200ms. Bei einer Rate von ca. 49 bearbeiteten Anfragen pro Sekunde wurden 6,5 Megabyte Daten bereitgestellt. Bei dem Test des apollo-commercetools-adapters stellte sich folgende Performance heraus.

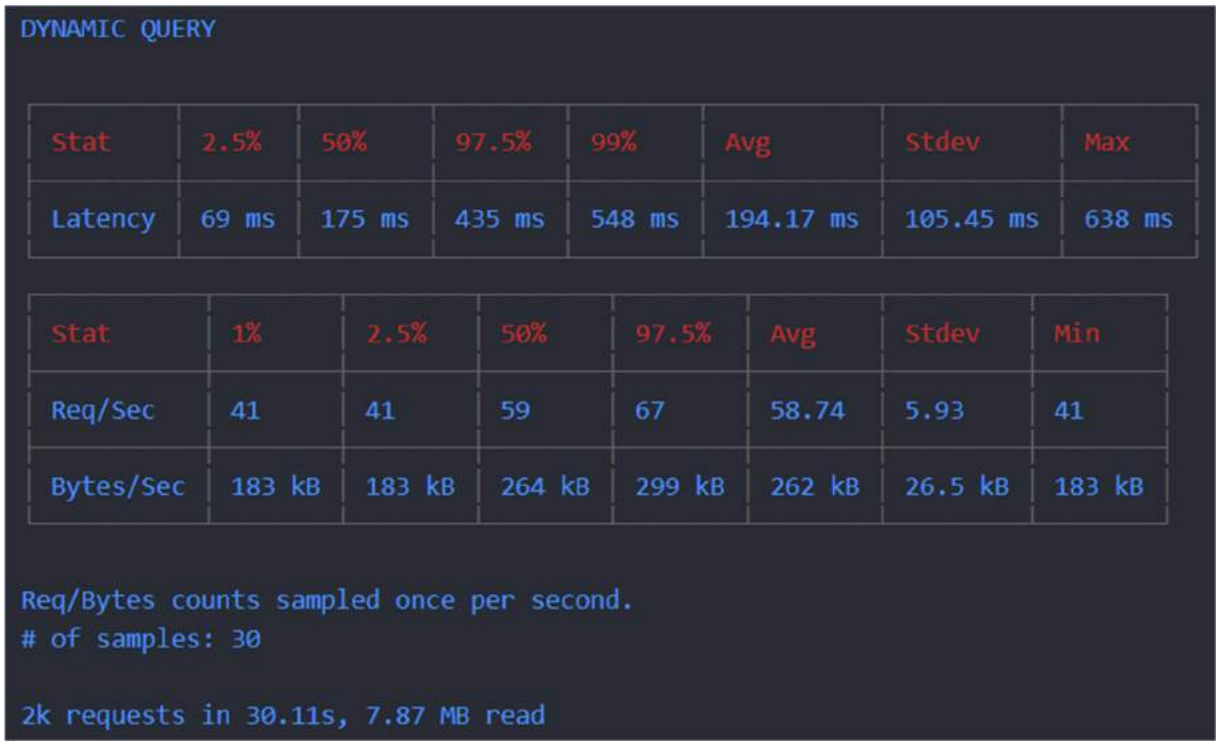


Abbildung 55: Performancetest mit dynamischem Schema

Mit einer durchschnittlichen Antwortzeit von ca. 194ms liegt die Latenz des apollo-commercetools-adapters nur geringfügig unter der des static-commercetools-adapters. Jedoch konnten vom apollo-commercetools mit 59 Anfragen pro Sekunde 7,9 Megabyte Daten in der gleichen Zeit zur Verfügung gestellt werden. Somit erweist der Ansatz der dynamischen Anfragen als performanter. Insbesondere, in realen Szenarien, in denen das Datenvolumen der Anfragen noch größer ist, gewinnt der Performancevorteil an Bedeutung.

Somit ist festzuhalten, dass die erste in Kapitel 4 formulierte Forschungsfrage hinsichtlich der Machbarkeit erfolgreich beantwortet wurde.

Die GraphQL-API von commercetools wurde unter den dort formulierten Bedingungen erfolgreich angebunden. Neben dynamischen Anfragen, können ebenfalls Variablen innerhalb der *Query* genutzt werden. Die Funktionalität war bisher im Rahmen der statischen Anfragen so nicht möglich. Weiterhin lässt sich auch ein Performancegewinn bei der dynamischen Lösung feststellen.

5.2 Auswertung der Integration in Apollo Federation

In Kapitel 4.3.1 wurde der apollo-commercetools-adapter so angepasst wurde, dass sein Schema oder genauer ein Typ des Schemas den Voraussetzungen von Apollo Federation entspricht und anschließend ein Apollo-Gateway aufgesetzt, um die Schemas des apollo-commercetools-adapters und des apollo-company-services zu vereinen. Um nun die zweite Forschungsfrage zu Integration in Apollo Federation aus Kapitel 4 zu beantworten, muss zunächst getestet werden, ob neben den Typen von commercetools auch der Typ *Company* aus dem apollo-company-service am Apollo-Gateway zur Verfügung stehen. Dazu kann dieses Mal die GraphQL-API des Apollo-Gateways mit einem Browser geöffnet werden, um das Schema in Apollo Studio zu betrachten.

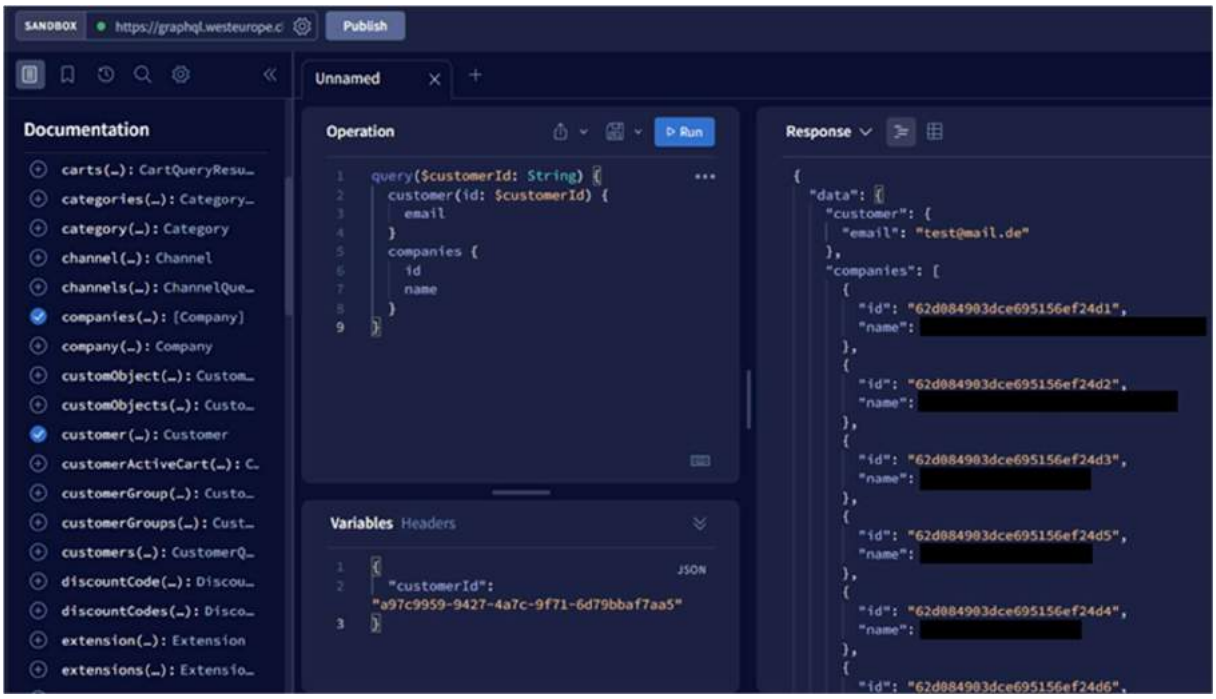


Abbildung 56: Apollo Studio - Kombinierte Query

Dieser Ausschnitt des Apollo Studio zeigt, dass ein *customer*-Objekt per ID und eine Liste aller *companies* innerhalb einer *Query* angefragt wurde. Die Liste der *companies* stammt dabei aus der MongoDB und wurde vom apollo-company-service bereitgestellt, das *customer*-Objekt stammt aus commercetools und wurde vom apollo-commercetools-adapter bereitgestellt. Das zeigt, dass die Typen von commercetools und des apollo-company-service im Apollo-Gateway parallel abgefragt werden können. Da weiterhin im Kapitel 4.3.2 der *Customer*-Typ um das Feld *company* erweitert wurde, kann nun gezeigt werden, dass die Services vollständig Federation unterstützen, wenn das neue Feld *company* im Typ *Customer* zur Verfügung steht und vom Apollo-Gateway korrekt aufgelöst werden kann.

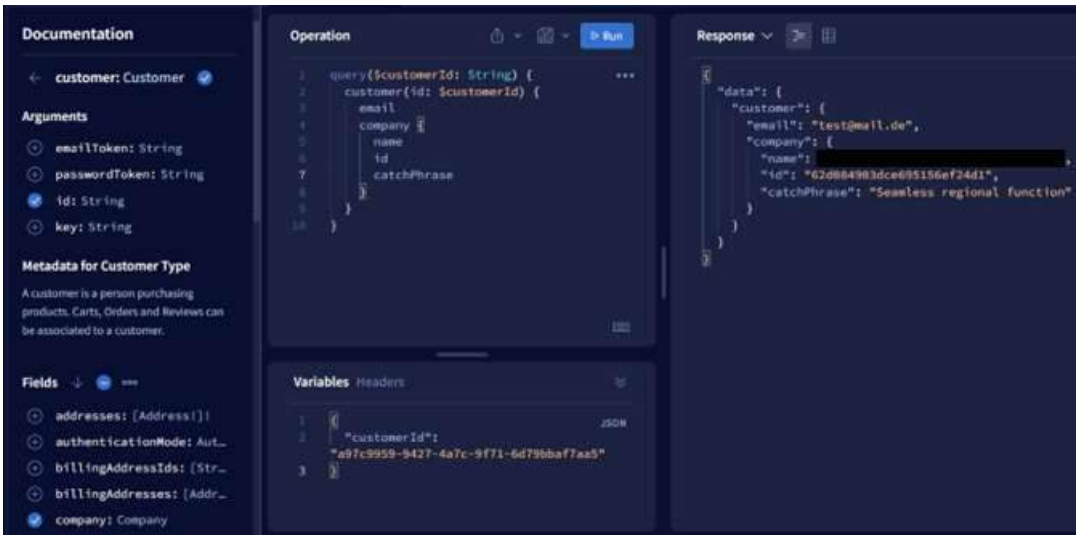


Abbildung 57: Apollo Federation - Einzelne Query

Dieser Ausschnitt des Apollo Studios zeigt auf der linken Seite, dass das Feld *company* nun im *Customer*-Typ korrekt im Schema des Apollo-Gateways angezeigt wird. Im mittleren Teil wurde ein *customer* über seine ID abgefragt. Dazu konnte jedoch nun auch zusätzlich seine zugehörige *company* direkt mit allen Attributen angefragt werden. Die rechte Seite des Ausschnittes, zeigt die Antwort bestehend aus dem *customer* von commercetools und der darin befindlichen *company* aus der MongoDB. Somit wurde gezeigt, dass die zweite Forschungsfrage aus Kapitel 4 erfolgreich umgesetzt wurde, da das Schema der GraphQL-API von commercetools mittels des apollo-commercetools-adapters vollständig Apollo Federation unterstützt. Auch die zusätzliche Bedingung, dass das Schema vollautomatisiert konvertiert werden soll, wurde erfüllt.

5.3 Auswertung der Lösung zum „N+1“-Problem

Um zu prüfen, ob die Implementierung des *dataloaders* zu einer Steigerung der Performance führt, wurden zwei Tests durchgeführt. Zuerst soll gezeigt werden, dass weniger Anfragen vom apollo-company-service an die MongoDB gesendet werden. Danach soll mit Hilfe eines Performancetests die Performance der Anfragen gemessen und festgestellt werden, ob ein *dataloader* dazu führt, dass mehr an Anfragen schneller bearbeitet werden können.

Um die eingehenden Anfragen an die MongoDB zu messen, wird das Programm „MongoDB Compass“ verwendet. Dieses kann mit einer Instanz einer MongoDB verbunden werden, um unter anderem die Performance dieser Datenbank auszuwerten. Dazu bietet das Programm unter dem Menüpunkt *Databases* einen Reiter *Performance*. Folgender Ausschnitt des Apollo Studio zeigt die Anfrage an das Apollo-Gateway, die für diesen Test genutzt wurde.

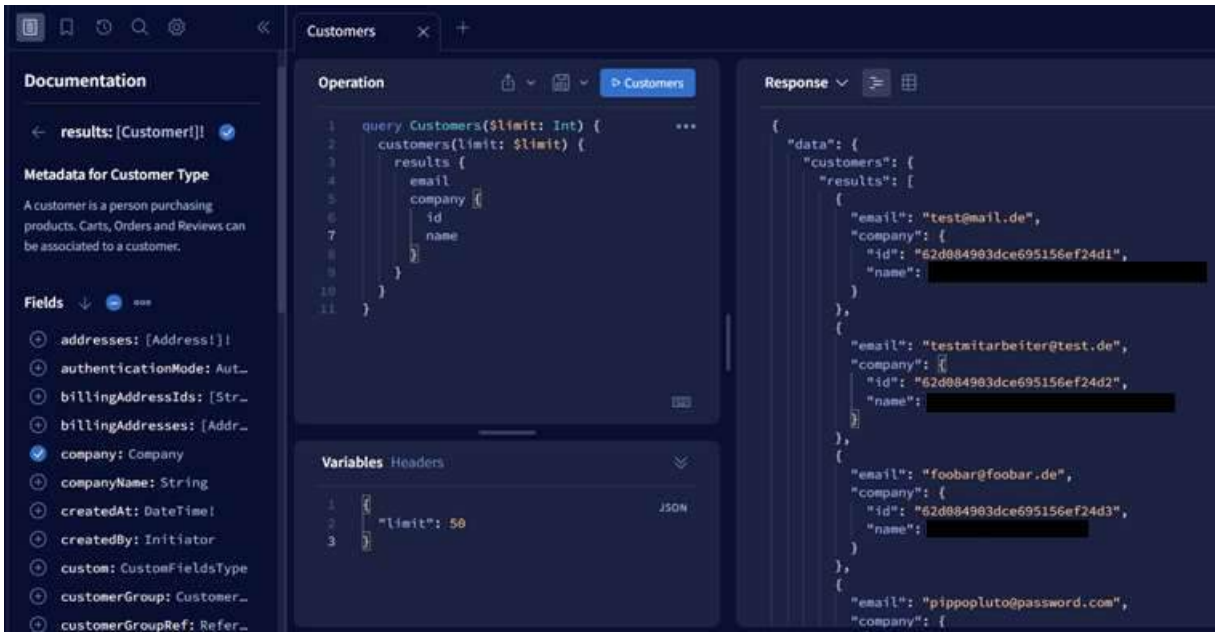


Abbildung 58: Dataloader - Query

In Abbildung 58 ist eine Anfrage einer Liste von *customer*-Objekten zu sehen. Dabei soll für jeden Kunden auch die Firmendaten abgefragt werden. Dabei wird ein Limit als Argument übergeben, welches unten bei den Variablen auf 50 gesetzt wurde. Daraus resultiert, wie bereits in Kapitel 3.2.3 angedeutet, dass alle 50 *customer*-Objekte in einer Anfrage vom apollo-commercetools-adapter angefragt werden können. Jedoch müssen zusätzlich für all diese Objekte die Firmendaten vom apollo-company-service und somit aus der MongoDB angefragt werden. Die Implementierung des folgenden Performancetests ist ebenfalls dem Anhang 3 zu entnehmen. Folgender Ausschnitt aus der Performanceübersicht von MongoDB Compass zeigt, dass tatsächlich 50 Anfragen bei der MongoDB eingegangen sind.

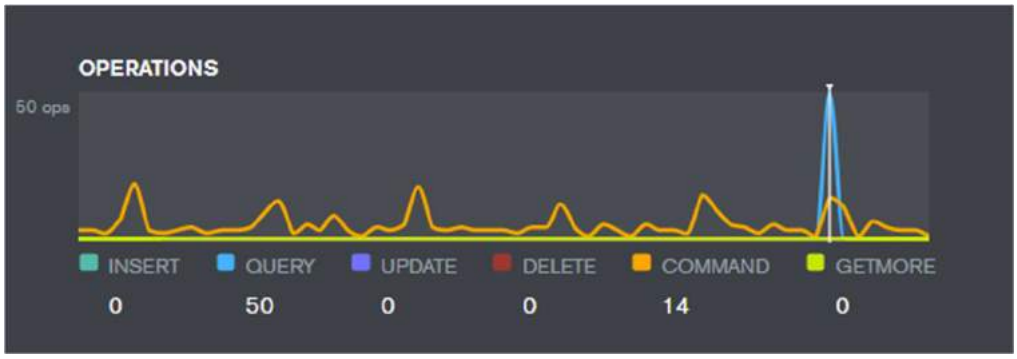


Abbildung 59: MongoDB Query - Ohne Dataloader

Der Cursor wurde auf den Ausschlag der blauen Verlaufslinie gelegt. Unter dem Diagramm ist zu sehen, dass 50 mal eine Query ausgeführt wurde. Die in Kapitel 4.4 implementierte Umgebungsvariable *USE_DATALOADER*, wurde anschließend auf *true* gestellt, um dieselbe Anfrage mit *dataloader* zu testen. Dabei ergab sich folgender Ausschnitt aus MongoDB Compass.

Abbildung 60: MongoDB Query - Mit Dataloader

Es ist also deutlich zu sehen, dass nun nur noch eine einige Anfrage an die MongoDB gestellt wurde.

Als nächstes wurde die Performance beider Varianten verglichen. Dazu wurde dieselbe Anfrage ebenfalls mit der Bibliothek *autocannon* für 30 Sekunden über 10 parallele Verbindungen so oft wie möglich an das Apollo-Gateway gesendet. Mit deaktiviertem *dataloader* ergab sich folgende Performance.

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	177 ms	310 ms	839 ms	1111 ms	341.82 ms	163.55 ms	1576 ms

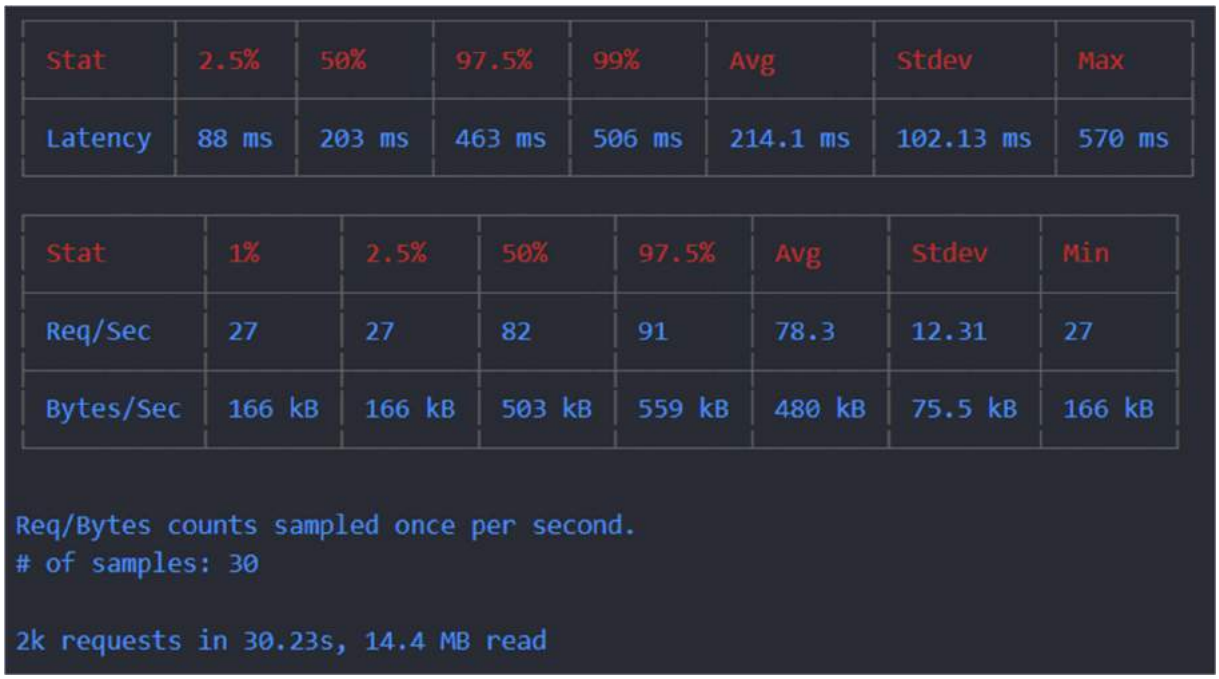
Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	3	3	32	39	29.27	8.81	3
Bytes/Sec	18.4 kB	18.4 kB	196 kB	239 kB	180 kB	54 kB	18.4 kB

```
Req/Bytes counts sampled once per second.
# of samples: 30

888 requests in 30.26s, 5.39 MB read
```

Abbildung 61: Performancetest - Ohne Dataloader

Abbildung 61 zeigt, dass 888 Anfragen bearbeitet werden konnten. Dabei wurden ca. 5,4 Megabyte Daten vom Apollo-Gateway bereitgestellt. Die durchschnittliche Latenz lag dabei bei ca. 340ms.



Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	88 ms	203 ms	463 ms	506 ms	214.1 ms	102.13 ms	570 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	27	27	82	91	78.3	12.31	27
Bytes/Sec	166 kB	166 kB	503 kB	559 kB	480 kB	75.5 kB	166 kB

Req/Bytes counts sampled once per second.
# of samples: 30

2k requests in 30.23s, 14.4 MB read

Abbildung 62: Performancetest - Mit Dataloader

In Abbildung 62 ist eine deutliche Performancesteigerung bei aktiviertem *dataloader* festzustellen. In diesem Fall konnte das Apollo-Gateway über zweitausend Anfragen bearbeiten und in der gleichen Zeit 14,4 Megabyte Daten bereitstellen. Dabei fällt die durchschnittliche Latenz mit ca. 214ms um 126ms deutlich geringer aus. Auch hier kann die letzte in Kapitel 4 formulierte Forschungsfrage somit positiv beantwortet werden. Es ist mit wenigen Zeilen Programmcode möglich, die Performance solcher in Apollo Federation häufig auftretenden Anfragen, deutlich zu steigern. Am Ende des Kapitels 3.2.3 zunächst war nicht klar, dass das *Batching* Zeit sparen kann. Da in der Zeit, in der die eingehenden Anfragen gesammelt werden, diese Anfragen blockiert sind und nicht bearbeitet werden können. Der Test zeigt jedoch, dass effektiv Zeit gespart wurde, da das bearbeitete Datenvolumen mehr als verdoppelt wurde.

6 Fazit

Insgesamt konnten, die in dieser Arbeit, gestellten Forschungsfragen hinsichtlich ihrer Umsetzbarkeit positiv beantwortet werden. Zusätzlich stellte sich heraus, dass die eingesetzten Lösungen einen messbaren Performancegewinn mitbringen. Die hier aufgezeigten Probleme stellen bislang große Hindernisse in Kundenprojekten dar. Somit ist GraphQL derzeit nur mit größerem Aufwand in Kundenprojekten nutzbar. Es wird viel Zeit investiert, um an vorübergehenden Lösungen, die GraphQL-API von commercetools mittels Apollo Federation in die eigene Architektur einzubinden, zu arbeiten. Ebenfalls ist die Wartung des aktuellen Standes mit all seinen statisch implementierten Anfragen und Schemas sehr aufwändig. Daher haben die hier gezeigten Probleme Potenzial in einem Kundenprojekt derartige Probleme zu beseitigen und somit den Einsatz von GraphQL effizienter zu gestalten. Vor allem in Anbetracht der Tatsache, dass sich diese Lösungen in der Testumgebung als performanter erwiesen haben. Trotzdem muss das Verhalten und die Performance der Lösungen in einem realen Szenario nochmals evaluiert werden. Obwohl die Testumgebung einem Ausschnitt aus einer realen Architektur sehr nahekommt, wird in Kundenprojekten mit anderen Datenvolumen und komplexeren Anfragen gearbeitet. Solche komplexen Szenarien können innerhalb dieser Arbeit nicht nachgestellt werden.

Außerdem muss ein weiterer sehr wichtiger Punkt betrachtet werden, bevor solch ein Lösung in einem Projekt produktiv eingesetzt werden kann. In Kapitel 4.3.1 wurde eine Bibliothek *graphql-transform-federation* zur Konvertierung von GraphQL Schemas verwendet. Diese Bibliothek ist veraltet und wird nicht mehr gewartet. Ohne eine nachträgliche Anpassung wäre diese Bibliothek nicht funktional gewesen. Jedoch liefert sie grundlegende Ansätze, um GraphQL Schemas zu konvertieren und sollte deshalb auch weiterhin dafür genutzt werden. Dafür muss diese Bibliothek vor einem produktiven Einsatz sehr stark überarbeitet und anschließend gewartet werden. Nur auf dieser Basis kann ein möglicher apollo-commercetools-adapter anschließend in Kundenprojekten effektiv genutzt werden.

Ein gänzlich anderer möglicher Ansatz die GraphQL-API von commercetools anzubinden könnten die Bibliotheken von *The Guild* sein. Dabei handelt es sich um eine Gruppe von Entwickler, die an Open-Source Bibliotheken für GraphQL arbeiten und diese der Community zur Verfügung stellen. Hierbei könnte ist besondere das Feature *GraphQL Mesh*

interessant sein. Jedoch müsste hier in weiteren Untersuchungen die Machbarkeit und Performance unter Verwendung dieser Bibliotheken evaluiert werden.

Aufbauend auf dieser Lösung könnte ebenfalls das Thema rund um Authentifizierung und Autorisierung untersucht werden. Diese Thematik wurde aufgrund ihres Umfanges nicht innerhalb dieser Arbeit betrachtet, spielt aber Kundenprojekten ebenfalls eine große Rolle. Hier ist im Speziellen interessant, wie der Zugriff auf einzelne Felder über das Apollo Gateway basierend Rechten eingeschränkt werden kann.

Aufgrund dessen, dass sich letzter Zeit alle GraphQL-Bibliotheken sehr schnell weiterentwickeln und es für einige Probleme noch keine etablierten Lösungsansätze gibt, kann festgehalten werden, dass es sich bei GraphQL um eine vergleichsweise junge Technologie handelt. Mit vorangeschrittenem Entwicklungsstand könnte es weitaus mehr Lösungsmöglichkeiten für die Probleme dieser Arbeit geben.

III Anhangsverzeichnis

Anhang 1 static-commercetools-adapter – Query

```
export default (limit=100) => {
    return `
    query {
      customers(limit: ${limit}) {
        results {
          customerNumber
          email
          defaultShippingAddressId
          defaultBillingAddressId
          shippingAddressIds
          billingAddressIds
          isEmailVerified
          externalId
          key
          authenticationMode
          firstName
          lastName
          title
          locale
          salutation
          dateOfBirth
          vatId
          password
          id
          version
          createdAt
          lastModifiedAt
          createdBy {
            externalUserId
            anonymousId
            clientId
          }
          lastModifiedBy {
            externalUserId
            anonymousId
            clientId
          }
        }
      }
    }`
}
```

Anhang 2 Performancetest - statisch vs. dynamisch

```
const autocannon = require('autocannon')
const printConfig = {outputStream: process.stdout}

const options = (path)=>({
  url: 'https://graphql.westeurope.cloudapp.azure.com',
  connections: 5,
  duration: 30,
  verifyBody: (body) => {
    return !body.errors
  },
  requests: [
      {
        method: 'POST',
        path: path,
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
            query: `{
                customers(limit: 100){
                  results {
                    id
                  }
                }
              }`
        })
      }
    ]
})

autocannon(options('/static/commercetools/graphql'), (err, result) => {
console.log("STATIC QUERY\n",autocannon.printResult(result, printConfig))
     autocannon(options('/apollo/commercetools/graphql'), (err, result) =>
{
console.log("DYNAMIC QUERY\n",autocannon.printResult(result, printConfig))
 })
})
```

Anhang 3 Performancetest „N+1“-Problem

```
const autocannon = require('autocannon')

const options = {
  url: 'https://graphql.westeurope.cloudapp.azure.com',
  connections: 10,
  duration: 30,
  requests: [
    {
      method: 'POST',
      path: '/apollo/graphql',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
          query: `{
              customers(limit: 50){
                results {
                  email
                  company {
                    id
                    name
                  }
                }
              }
            }`
      })
    }
  ]
}

autocannon(options, (err, result) => {
    console.log(autocannon.printResult(result, {outputStream:
process.stdout}))
})
```

IV Literaturverzeichnis

[Apo 22a] Apollo Graph Inc.: „Apollo Docs Home“, 2022.
https://www.apollographql.com/docs/
Abruf: 2022.07.28

[Apo 22b] Apollo Graph Inc.: „Entities in Apollo Federation“, 2022.
https://www.apollographql.com/docs/federation/entities
Abruf: 2022.07.28

[Apo 22c] Apollo Graph Inc.: „Federation-specific GraphQL directives“, 2022.
https://www.apollographql.com/docs/federation/federated-types/federated-directives
Abruf: 2022.07.28

[Apo 22d] Apollo Graph Inc.: „GraphQL query best practices“, 2022.
https://www.apollographql.com/docs/react/data/operation-best-practices/
Abruf: 2022.07.28

[Apo 22e] Apollo Graph Inc.: „Implementing subgraphs“, 2022.
https://www.apollographql.com/docs/federation/subgraphs
Abruf: 2022.07.28

[Apo 22f] Apollo Graph Inc.: „Implementing the gateway“, 2022.
https://www.apollographql.com/docs/federation/gateway
Abruf: 2022.07.28

[Apo 22g] Apollo Graph Inc.: „Introduction to Apollo Studio“, 2022.
https://www.apollographql.com/docs/studio
Abruf: 2022.07.28

[Byr 15] Byron, Lee: „GraphQL: A data query language“, 2015.
https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/
Abruf: 2022.07.28

[com 22a] commercetools GmbH: „Authorization“, 2022.
https://docs.commercetools.com/api/authorization
Abruf: 2022.08.01

[com 22b] commercetools GmbH: „TypeScript SDK Overview“, 2022.
https://docs.commercetools.com/sdk/javascript-sdk
Abruf: 2022.08.01

[Fielding et al. 04a] Fielding, R.; Reschke, J.: „RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content“, 2004.
https://datatracker.ietf.org/doc/html/rfc7231#section-4.3
Abruf: 2022.07.29

[Fielding et al. 04b] Fielding, R.; Reschke, J.: „RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content“, 2004.
https://datatracker.ietf.org/doc/html/rfc7231#section-6
Abruf: 2022.07.29

[Jak 05] Jaki, Michael: „Representational State Transfer“, 2005.
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.7334&rep=rep1&type=pdf
Abruf: 2022.07.29

[Mukhiya et al. 19] Mukhiya, S., et al.: „A GraphQL approach to Healthcare Information Exchange with HL7 FHIR“, in Procedia Computer Science, 2019. S. 338–345

[Oos 22] Oostinga, Ruben: „0xR/graphql-transform-federation: Convert your existing GraphQL schema into a federated schema“.
https://github.com/0xR/graphql-transform-federation
Abruf: 2022.08.01

[Stu 17] Sturgeon, Phil: „GraphQL vs REST: Overview“, 2017.
https://phil.tech/2017/graphql-vs-rest-overview/
Abruf: 2022.07.28

[The 22a] The GraphQL Foundation: „Execution | GraphQL“, 2022.
https://graphql.org/learn/execution/
Abruf: 2022.07.28

[The 22b] The GraphQL Foundation: „GraphQL | A query language for your API“, 2022.
https://graphql.org/
Abruf: 2022.07.28

[The 22c] The GraphQL Foundation: „GraphQL Best Practices | GraphQL“, 2022.
https://graphql.org/learn/best-practices/
Abruf: 2022.07.29

[The 22d] The GraphQL Foundation: „Introduction to GraphQL | GraphQL“, 2022.
https://graphql.org/learn/
Abruf: 2022.07.28

[The 22e] The GraphQL Foundation: „Introspection | GraphQL“, 2022.
https://graphql.org/learn/introspection/
Abruf: 2022.07.28

[The 22f] The GraphQL Foundation: „Queries and Mutations | GraphQL“, 2022.
https://graphql.org/learn/queries/
Abruf: 2022.07.28

[The 22g] The GraphQL Foundation: „Serving over HTTP | GraphQL“, 2022.
https://graphql.org/learn/serving-over-http/
Abruf: 2022.07.29

[The 22h] The GraphQL Foundation: „Who's Using | GraphQL“, 2022.
https://graphql.org/users/
Abruf: 2022.07.28

V Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Bachelorarbeit mit dem Thema:

   „GraphQL in einer Microservicearchitektur“

   ohne fremde Hilfe angefertigt habe,
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und
3. dass ich meine Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum [redacted]