GraphQL als Alternative zu REST

Manuel Mauky, Saxonia Systems AG

GraphQL ist eine Abfragesprache für Web-APIs, mit der Web- und Mobile-Anwendungen Daten vom Server beziehen könnten. Es stellt damit eine Alternative zu REST-Schnittstellen dar, die in diesem Kontext häufig eingesetzt werden. Der Artikel zeigt, wie GraphQL einige der Schwierigkeiten von REST in diesem Einsatz-Szenario löst und darüber hinaus interessante neue Ideen bei der Frontend-Entwicklung ermöglicht.

Der Aufbau und die Architektur vieler Anwendungsarten hat sich in den letzten Jahren in mehrfacher Hinsicht gewandelt. Auf der Backend-Seite findet man heute beispielsweise immer häufiger auf Modularität ausgerichtete Ansätze wie Microservices vor. Auch die Frontend-Seite hat sich gewandelt. Während früher das Rendering größtenteils vom Server übernommen wurde und der Client lediglich einfache Hilfsaufgaben übernommen hat, sind heute komplexe Single-Page-Anwendungen weitverbreitet. Diese Entwicklung ist für viele Anwendungsfälle auch sinnvoll, da damit eben auch komplexere Anwendungen mit höheren Anforderungen an das Interaktionsdesign realisiert werden können.

Eine ähnliche Entwicklung konnte man auf mobilen Geräten beobachten, bei denen heute immer mehr Anwendungen durch eigenständige Apps umgesetzt werden. Hieraus ergibt sich eine klare Aufteilung der Zuständigkeiten zwischen dem Server und den Client-Anwendungen. Der Server stellt Daten bereit und bietet Schnittstellen für fachliche Aktionen. Von der Interaktion mit dem Nutzer hingegen hat der Server keine Kenntnis.

Für Entwickler bleibt die Frage, wie konkret nun die Daten vom Server abgeholt, zum Client transportiert und dort verarbeitet werden können. Die klassische Antwort lautet „REST". Doch obwohl REST mittlerweile omnipräsent ist und auch in sehr vielen anderen Szenarien wie zur Kommunikation zwischen Microservices erfolgreich zum Einsatz kommt, ist es nicht unbedingt immer die beste Wahl. Einer dieser Anwendungsfälle, für die REST zwar gegenwärtig häufig zum Einsatz kommt, aber gleichzeitig für einige Probleme und so manches Kopfzerbrechen sorgt, ist der Datenaustausch zwischen dem Server und Client-Anwendungen. Genau aus diesem Grund und für diesen Einsatzzweck hat Facebook die Abfragesprache „GraphQL" entwickelt, um einige Probleme von REST zu lösen. Dieser Artikel stellt deshalb GraphQL als Alternative zu REST für derartige Szenarien vor.

REST ist Ressourcen-orientiert. Jede Ressource besitzt einen eindeutigen Identifier, den Unified Resource Identifier (URI). Unter diesem können Repräsentationen dieser Ressource, beispielsweise als JSON oder XML, abgeholt werden. Ein weiteres Merkmal von REST ist die Verwendung von Hyperlinks zwischen Ressourcen. Damit lassen sich unter anderem Verbindungen und Abhängigkeiten zwischen Daten ausdrücken.

Als Beispiel dient eine Blog-Anwendung. Diese könnte die Ressourcen „Article", „Author" und „Comment" besitzen. Ein Artikel wurde von einem oder mehreren Autoren verfasst und kann Kommentare enthalten. Als Besucher des Web-Blogs landet man zunächst auf der Artikel-Übersichtsseite. Sie zeigt die letzten zehn Artikel in einer Kurz-Ansicht mit den Namen der Autoren und der Anzahl der Kommentare. Anschließend kann man über einen Klick auf eine Überschrift zu einer Detailseite des jeweiligen Artikels gelangen.

Aus Frontend-Sicht beginnt die Kommunikation mit einem GET-Request auf den Einstiegspunkt des API, woraufhin eine Liste der verfügbaren Ressourcen geliefert wird. Anschließend liefert ein weiterer GET-Request auf die „Article"-Ressource eine Liste von Artikeln. Hierbei stößt man auf die erste Schwierigkeit, denn es sollen ja nur die letzten zehn Artikel abgefragt werden. REST bietet aber kein Konzept für die Sortierung und Begrenzung der gelieferten Daten. Allerdings existieren Best-Practices und Lösungsansätze zur Gestaltung des API, um auch diese Funktionalitäten anbieten zu können.

```
type Article {
  id: ID!
  releaseDate: String
  teaser: String
  text: String
  permalink: String
  authors: [Author]
  comments: [Comment]
}

type Author {
  id: ID!
  name: String!
  articles: [Article]
}

type Comment {
  id: ID!
  text: String!
  author: Author
]
```

Listing 1

```
{
  Article {
    id
    title
    permalink
    teaser
    releaseDate
    authors {
       name
    }
    comments {
       id
    }
  }
}
```

Listing 2

Nun hat das Frontend zwar eine Liste der Artikel, es fehlen jedoch noch bestimmte Informationen zur Anzeige. Da Autoren und Kommentare jeweils eigene Ressourcen darstellen, beinhalten die ausgelieferten Artikel-Daten diese Informationen nicht selbst, sondern verweisen lediglich per Hyperlink auf die verbundenen Ressourcen. Um an die eigentlichen Daten zu gelangen, sind für jeden Artikel in der Liste zwei weitere GET-Requests auf die jeweilige Autoren- beziehungsweise Kommentar-Ressource durchzuführen.

Für dieses einfache Beispiel der Artikel-Übersichtsseite sind also bereits 22 GET-Requests erforderlich. Dieser Umstand kann ein ernstes Performance-Problem darstellen, da die Anzahl notwendiger Requests ein wesentlicher Faktor für die Geschwindigkeit der Anwendung sein kann. Das ist insbesondere der Fall für mobile Anwendungen, die auch in Gegenden mit schlechter Netzabdeckung noch benutzbar sein sollen.

Die Anzahl der Requests ist jedoch nicht das einzige Problem. Ein weiteres ist die Menge der zu übertragenden Daten. Im Beispiel soll ein Artikel ein Feld für den eigentlichen Text und eines für einen kurzen Teaser enthalten. Für die Startseite ist nur der Teaser inte-

```
{
  "data": {
    "Article": [
      {
        "id": "sd3423s32",
        "title": "Some Title",
        "permalink": "some_title",
        "teaser": "Lorem Ipsum...",
        "releaseDate": "2017-04-23",
        "authors": [
           {
              "name": "Hugo"
           }
        ],
        "comments": []
      },
      {
        "id": "sdöf234ds2",
        "title": "Other Title",
        "permalink": "other_title",
        "teaser": "Lorem Ipsuns.sd",
        "releaseDate": "2017-03-24",
        "authors": [
           {
              "name": "Luise"
           },
           {
              "name": "Elfriede"
           }
        ],
        "comments": [
            {
              "id": "13424"
            }
        ]
      }
    ]
  }
}
```

Listing 3

ressant, der ausführliche Text soll dagegen erst auf der Detailseite erscheinen. Je nachdem, wie das REST-API gestaltet ist, könnte es allerdings sein, dass auch der ausführliche Text bereits bei dem Request für die Artikel-Liste für jeden dieser Artikel mitgeliefert wird und damit unnötige Daten übertragen werden. Als Entwickler wünscht man sich also auch hier eine Möglichkeit, um das Ergebnis eines REST-Requests zu beeinflussen.

Auch für diese Probleme gibt es Lösungsansätze bei REST. Beispielsweise könnten Query-Parameter mitgeschickt werden, die bestimmen, welche Daten zurückgeliefert werden. Wenn man beispielsweise den Parameter „withFullText=true" an die URL anhängt, könnte der Server entsprechend den vollständigen Text für jeden Artikel mit in die Antwort packen. Eine weitere Option wäre, verschiedene Repräsentationen für ein und dieselbe Ressource bereitzustellen. Bei einem GET-Request könnte dann etwa als ACCEPT-Header nicht mehr nur „application/json" sondern „application/full+json" oder „application/with.comments+json" angegeben werden.

Allerdings lösen beide Ansätze nicht das Problem, dass für eine Ansicht manchmal Daten von mehreren Ressourcen benötigt werden. Um diesem zu begegnen, sieht man deshalb in der Praxis manchmal View-spezifische Ressourcen. Ein GET-Request auf „/api/article_overview" könnte beispielsweise die Kurzfassung der Artikel liefern, genauso wie es für die Übersichtsseite notwendig ist. Hierbei werden allerdings die Ideen von REST ziemlich verwässert. Denn eigentlich ist „article_overview" keine eigenständige, echte Ressource. Vor allem sorgt dieser Ansatz aber für eine deutliche Kopplung zwischen Server und Clients. Ist eine neue Ansicht im Frontend geplant, müssen dafür entsprechende Änderungen am Server vorgenommen werden, selbst dann, wenn sich die eigentlichen Daten gar nicht geändert haben.

Wie man es also dreht und wendet, eine wirklich zufriedenstellende Lösung scheint unmöglich zu sein. Will man ein sauberes REST-API umsetzen, erschwert dies die Benutzung auf Frontend-Seite. Weicht man dagegen die Regeln für REST auf und verzichtet beispielsweise auf saubere Hyperlinks zwischen den Ressourcen, muss man eine stärkere Kopplung zwischen Server und Clients in Kauf nehmen. Selbst dann ist die Benutzung des API auf Frontend-Seite alles andere als optimal, da immer noch überflüssige Daten übertragen werden können.

GraphQL als Alternative

Basierend auf diesen Schwierigkeiten beim Umgang mit REST-Schnittstellen für die Abfrage von Daten und den Transport zum Client, hat Facebook die Abfragesprache GraphQL entwickelt. Sie ist dort seit dem Jahr 2012 intern im Einsatz und wurde Ende 2015 der Allgemeinheit präsentiert und zur Verfügung gestellt. Seitdem hat GraphQL vor allem in der React-Community, aber auch darüber hinaus für großes Interesse gesorgt. Beispielsweise hat GitHub seine neue Version des offiziellen API erst kürzlich auf GraphQL umgestellt *[1]* und verarbeitet darüber laut eigenen Angaben täglich mehr als 125 Millionen Queries.

Doch was ist GraphQL nun genau und wie hilft es bei den oben beschriebenen Problemen? Bei GraphQL stellt der Server ein statisch typisiertes Schema bereit. In diesem Schema ist festgelegt, welche Daten der Server kennt und wie diese Daten miteinander in Beziehung stehen. In *Listing 1* ist beispielhaft ein Ausschnitt aus einem Schema für unseren Blog-Anwendungsfall zu sehen. Es enthält für unsere drei Entitäten „Author", „Article" und „Comment" Typdefinitionen.

GraphQL selbst stellt zunächst einmal nur eine Spezifikation dar. Die im Code-Ausschnitt verwendete Syntax ist die GraphQL Schema Language *[2]*; sie dient vor allem der Beschreibung des Schemas in einer Programmiersprachen-unabhängigen Art und Weise. Wie das Schema konkret implementiert wird, hängt von der verwendeten Programmiersprache ab. Hier macht GraphQL selbst keine Vorgaben, die Macher stellen aber eine Referenz-Implementierung in JavaScript zur Verfügung. Vom Frontend aus können nun Queries gegen dieses Schema geschrieben und zum Server geschickt werden. *Listing 2* zeigt eine solche Beispiel-Query für unsere Artikel-Übersichtsseite.

```
// schema
type Mutation {
    addAuthor(name: String): Author
}

// query
mutation {
    addAuthor("Manuel") {
        id
    }
}
```

Listing 4

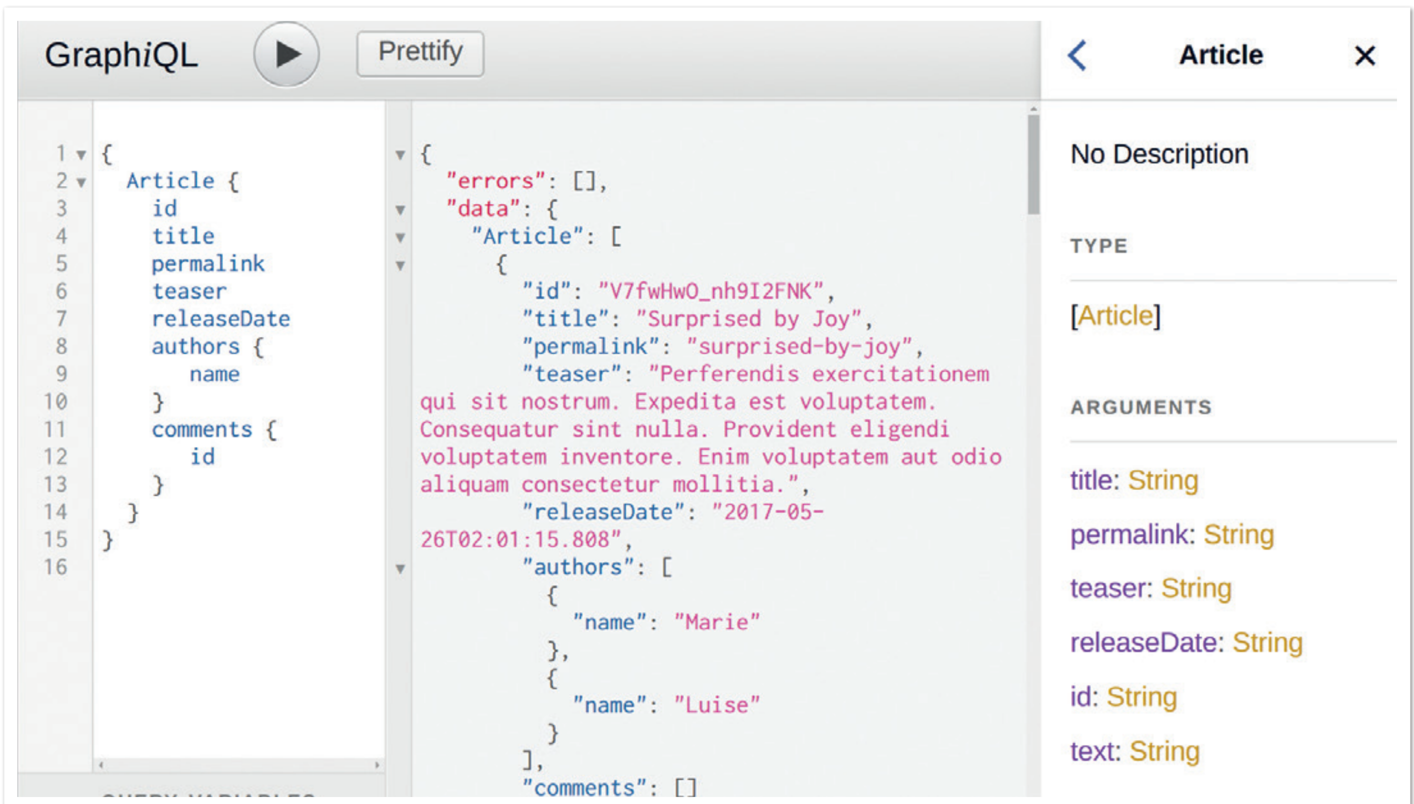

Abbildung 1: Das Tool GraphiQL zum Ausprobieren von APIs

Die Query erinnert in ihrer Struktur ein wenig an JSON. Sie beschreibt genau, welche Daten erforderlich sind. Wir interessieren uns für Artikel. Von diesen interessieren uns aber nur die aufgeführten Attribute. Außerdem können Abfragen beliebig geschachtelt werden. Dies machen wir uns für die Autoren des Artikels zunutze, da wir von den Autoren für die Übersichtsseite lediglich die Namen benötigen. Auch hier können wir also wieder genau bestimmen, welche Attribute von Interesse sind und welche nicht. *Listing 3* zeigt eine Antwort, wenn man diese Query zum Server schickt.

Die Antwort liegt als JSON vor und entspricht in ihrer Struktur genau der gestellten Anfrage. Nutzer des API können also exakt bestimmen, welche der angebotenen Daten sie interessieren und welche nicht. Wie der Server die Daten für die jeweilige Query beschafft, hängt von der Implementierung ab. Auf unterster Ebene bietet GraphQL Einstiegspunkte auf jeder Verschachtelungsebene des Schemas. Bei der Verarbeitung der Query aus *Listing 2* entspricht beispielsweise „Article" letztlich einem Funktionsaufruf, mit dem alle Artikel aus der Datenbank oder aus einer sonstigen Quelle geladen werden müssen. Auf der nächsten Ebene entsprechen wiederum alle Felder, also „id", „title" etc., letztlich Funktionsaufrufen, die für einen gegebenen Artikel einen konkreten Wert liefern müssen. Für skalare Typen wird hier standardmäßig einfach ein „Getter" aufgerufen, es können jedoch auch andere Implementierungen gewählt werden.

Die Varianten, um ein GraphQL-API zu implementieren, sind vielfältig. Für Java-Entwickler existiert zum Beispiel eine von der Community entwickelte Bibliothek *[3]*, um aus vorhandenen JPA-Entitäten automatisch ein passendes GraphQL-Schema zu generieren. Dabei wird auch das Beschaffen der Daten über den EntityManager von JPA automatisch von der Bibliothek übernommen.

Mutationen

Bisher wurde nur die Abfrage von Daten betrachtet. REST stellt aber auch Möglichkeiten für deren Manipulation und Veränderung bereit. Dazu werden bei REST die wohldefinierten HTTP-Verben verwendet. Ein POST-Request auf eine Ressource sorgt beispielsweise für das Anlegen einer neuen Entität, ein PUT-Request für das Editieren und ein DELETE-Request für das Löschen einer Ressource, wobei das REST-API definiert, welche Operationen für welche Ressourcen erlaubt sind.

Auch GraphQL erlaubt verändernde Operationen, jedoch unterscheidet sich der Ansatz ebenfalls fundamental von REST. Die möglichen Operationen sind bei REST durch die vorhandenen Verben vorgegeben und fest mit den Ressourcen verknüpft. Bei GraphQL werden dagegen sogenannte „Mutationen" im Schema definiert, die allerdings vollkommen losgelöst von der Abfrageseite sind.

Mutationen sind im Prinzip als ausführbare fachliche Funktionen zu verstehen, die Daten übergeben bekommen und Daten zurückliefern können, wobei üblicherweise die veränderten Daten selbst als Rückgabewerte angeboten werden. In *Listing 4* ist eine Mutation „addAuthor" definiert, die zum Anlegen von neuen Autoren benutzt werden kann. Zusätzlich zum übergebenen Namen lässt sich eine GraphQL-Query angeben, die im Beispiel die ID des neu angelegten Autors selektiert.

Ohne es explizit zu erwähnen, setzt die GraphQL-Spezifikation also im Prinzip das „Command-Query-Responsibility-Segregation"-Pattern (CQRS) um. Es wird strikt zwischen Abfrage und Veränderung der Daten getrennt. Auf diese Weise lassen sich sehr einfach auch fachliche Intentionen im API ausdrücken, indem zum Beispiel Mutationen wie „editText" und „changeAuthor" angeboten werden.

Obwohl beide letztlich den Artikel verändern, stecken doch unterschiedliche Absichten dahinter, die vom Server auch unterschiedlich behandelt werden könnten.

Statische Typisierung und Introspektion

GraphQL erlaubt nicht nur das Abfragen von konkreten Daten, sondern auch von Informationen über das Schema selbst. Diese als „Introspektion" bezeichnete Funktion sowie die Tatsache, dass GraphQL statisch typisiert ist, ermöglichen interessante und mächtige Entwicklertools zur Unterstützung der Programmierer. Das beste Beispiel dafür ist das Abfrage-Tool „GraphiQL" – man beachte das kleine „i" im Namen *[4] (siehe Abbildung 1)*. Es ermöglicht nicht nur das dynamische Ausführen von Queries gegen ein Schema, sondern unterstützt auch durch Autovervollständigung, automatische Hinweise beim Schreiben von Queries sowie durch eine automatisch bereitgestellte Dokumentation des Schemas. Auch weitere Anwendungsfälle sind denkbar. So könnte man beispielsweise bereits zum Entwicklungszeitpunkt durch ein in den Build-Prozess integriertes Tool prüfen, ob die im Client verwendeten Queries auch wirklich zum Schema des Servers passen, und bei Fehlschlägen frühzeitig Warnungen ausgeben *[5]*.

Aber auch die andere Richtung ist möglich: Ein Tool im Build-Prozess könnte prüfen, ob eine Änderung im Server-Code versehentlich zu einer inkompatiblen Änderung des API führt, indem alle vorhandenen Queries aller Client-Anwendungen automatisch gegenüber dem neuen API geprüft werden *[6]*.

Neue Ideen und Patterns im Frontend

Neben der Möglichkeit zur Vermeidung von unnötigen Requests durch zielgenau formulierte Queries ermöglicht GraphQL auch einige neue Ideen und Patterns bei der Frontend-Entwicklung. Diese resultieren nicht nur in einer weiteren Optimierung der zu übertragenden Daten, sondern ermöglichen darüber hinaus auch eine bessere Entkopplung von UI-Komponenten.

Dazu nochmal ein Blick auf das Blog-Beispiel. Für die Übersichtsseite sind einige, aber nicht alle Daten der Artikel erforderlich und man formuliert eine entsprechende GraphQL-Query. Bei einem Klick des Nutzers auf einen konkreten Artikel sollen zu einer Detail-Seite navigiert und dazu weitere Daten zu diesem konkreten Artikel nachgeladen werden, was wiederum durch eine entsprechende GraphQL-Query ausgedrückt wird. Die Menge der benötigten Daten überschneidet sich jedoch bei beiden Seiten. Informationen wie der Titel, der Teaser und die Namen der Autoren werden auf beiden Seiten benötigt. Andere Daten, vor allem der ausführliche Artikeltext, sind nur für die Detail-Seite relevant.

Beim Betreten der Detail-Seite liegen also bestimmte Daten bereits vor und müssen nicht erneut vom Server geladen werden. Als Entwickler könnte man diese Überlegung also nutzen, die Query für die Detailseite entsprechend kürzen und Daten von der Übersichtsseite wiederverwenden. Dieser naive Ansatz birgt allerdings einige Nachteile: Die Detailseite besitzt dann schließlich eine direkte Abhängigkeit zur Übersichtsseite. Ändert sich später die Darstellung auf der Übersichtsseite, könnten plötzlich bestimmte Informationen nicht mehr zur Verfügung stehen. Außerdem ist die Detailseite auch direkt über eine URL ansprechbar. In diesem Fall war die Übersichtsseite nicht vorher aktiv und hat noch keine Daten geladen. Ein händisches Optimieren nach diesem Muster ist also kompliziert und fehleranfällig.

Jedoch könnte ein entsprechend intelligentes Framework diese Optimierung im Hintergrund selbstständig übernehmen. Entwickler schreiben dann für jede Komponente eine komplette GraphQL-Query mit sämtlichen benötigten Daten und sind damit vollständig entkoppelt von anderen Komponenten. Erst das Framework übernimmt die Optimierung der Queries und setzt darüber hinaus ein Caching im Client um. Auf diese Weise müssen bestimmte Queries unter Umständen überhaupt nicht mehr neu ausgeführt werden, wenn beispielsweise später zu einer Seite zurücknavigiert wird. Der Umfang der zu übertragenden Daten verringert sich dadurch noch weiter, vor allem aber wird ein vergleichsweise einfaches Programmiermodell mit sehr starker Entkopplung ermöglicht: Entwickler einer Komponente definieren einfach ihre Datenanforderung mittels GraphQL, ohne sich Gedanken über das konkrete Laden der Daten und Aspekte wie Caching Gedanken machen zu müssen. Ein weiterer Vorteil dieser als „Query Co-Location" bezeichneten Methode ist, dass so auch verschachtelte Komponenten besser voneinander entkoppelt werden können.

Die Detail-Seite könnte beispielsweise aus einer Titel-Komponente, die den Titel des Artikels sowie die Informationen zum Autor darstellt, einer Haupt-Komponente, die den vollständigen Text des Artikels anzeigt, und einer Kommentar-Komponente zusammengesetzt sein. Jede dieser Unter-Komponenten definiert ihre jeweils benötigten Datenanforderungen mittels GraphQL. Es wäre allerdings eine schlechte Idee, wenn jede Komponente auch selbst dafür verantwortlich wäre, ihre Query auch tatsächlich auszuführen.

Stattdessen könnte jede Komponente ihre Query an die jeweilige Eltern-Komponente weiterreichen. Diese könnte die Queries der Kind-Komponenten kombinieren und ihrerseits wieder an die eigene Eltern-Komponente übergeben. Irgendwann, an der Wurzel des Komponentenbaums angelangt, müsste die vollständige Query einmalig ausgeführt und die Ergebnis-Daten wiederum auf gleichem Weg zurück in die Komponenten durchgereicht werden.

Dieser Mechanismus sorgt für eine weitere Entkopplung der UI-Komponenten, denn Eltern-Komponenten haben keine konkrete statische Abhängigkeit mehr zu den Datenanforderungen ihrer Kind-Komponenten. Sie sammeln lediglich die Queries ein, ohne Kenntnis davon, was in der Query jeweils steht. Benötigt eine Kind-Komponente beispielsweise wegen eines neuen Features neue Daten, sind in der Eltern-Komponente keinerlei Anpassungen im Code notwendig.

Interessant ist dabei auch, dass beim Kombinieren der einzelnen Queries durchaus auch Dopplungen enthalten sein können. Mehrere Komponenten könnten die gleichen Daten benötigen. Auch hier könnte ein intelligentes Framework also Optimierungen durchführen. Tatsächlich existieren mit „Relay" *[7]* und „Apollo Client" *[8]* mindestens zwei JavaScript-Frameworks, die solche Ansätze unterstützen. Relay wurde von Facebook selbst als umfassendes Framework für die Kombination von GraphQL mit React.js speziell für datengetriebene Anwendungen entwickelt. Apollo Client dagegen ist ein etwas leichtgewichtigerer Ansatz, der neben React.js auch andere Plattformen wie Angular, iOS und Android unterstützt.

Nachteile von GraphQL und Vorteile von REST

Natürlich gibt es auch bei GraphQL nicht nur Vorteile, sondern auch Nachteile und Einschränkungen. Und auch REST hat nach wie vor seine Berechtigung und in vielen Situationen Vorteile gegenüber GraphQL. Da wären zum einen ganz praktische Aspekte, wie die weite Verbreitung von REST, die damit verbundene Akzeptanz und die Verfügbarkeit von Wissen und Werkzeugen. Ein Beispiel ist das Spring-Data-REST-Projekt, mit dem Java-Entwickler extrem einfach REST-APIs aufsetzen können. Im GraphQL-Umfeld existieren zwar auch erste Projekte, die etwa eine Integration mit Spring-Boot und JPA bieten, aber bezüglich Ausgereiftheit, Stabilität und Dokumentation können diese Community-Ansätze noch nicht mithalten.

REST hat jedoch auch inhärente Vorteile, vor allem dann, wenn der Hypermedia-Gedanke konsequent verfolgt wird. So können beispielsweise mittels Hyperlinks ohne Weiteres auch Verbindungen zu ansonsten unabhängigen Ressourcen auch auf fremden APIs hergestellt werden, was mit GraphQL aufgrund des Fokus auf ein einziges Schema nicht so einfach möglich ist. Dies ist besonders im Microservices-Umfeld interessant, wo häufig ein API-Gateway einkommende Requests auf unterschiedliche Microservices umleitet und dieser Prozess auch anhand der Hyperlinks gesteuert werden kann.

Da die Beschaffung der Daten auch bei einem GraphQL-Schema letztlich frei implementiert werden kann, sind ähnliche Weiterleitungen auch mit GraphQL umsetzbar, allerdings ist der nötige Aufwand etwas höher. Ein weiterer Aspekt ist die Steuerung von Clients von der Serverseite aus durch intelligent gesetzte Hyperlinks. Dadurch kann das Verhalten der Clients manipuliert werden, ohne dass Anpassungen an der eigentlichen Software der Clients notwendig sind.

Ein Nachteil bei GraphQL ist, dass Requests nicht ohne Weiteres auf der Netzwerk-Ebene gecacht werden können. In vielen Fällen dürfte das allerdings wenig problematisch sein, da über das GraphQL-API in der Regel ja vor allem dynamische Daten abgefragt werden und diese im Vergleich zu statischen Inhalten wie HTML-, JavaScript- und Bild-Dateien, die weiterhin ganz normal cachbar sind, häufig einen geringen Umfang haben dürften.

Ein weiterer Nachteil ist, dass die Größe der Requests selbst, also der Daten, die vom Client zum Server geschickt werden müssen, in der Regel größer ist. Auch hier sind ebenfalls Optimierungen denkbar, die beispielsweise mit der aktuellen Version des oben erwähnten Relay-Frameworks eingeführt wurden, diese sind aber Framework-spezifisch.

Fazit

Als Fazit lässt sich sagen, dass GraphQL eine wirklich spannende Technologie mit interessanten Ideen ist und in einigen Anwendungsfällen eine echte Alternative zu REST-Schnittstellen darstellt. REST wird dadurch aber nicht obsolet, sondern behält seine Berechtigung durch seine Flexibilität und eine Vielzahl von Einsatzmöglichkeiten weiterhin bei.

Vor allem dann, wenn die Menge der zu übertragenden Daten und die Anzahl der Requests kritisch ist, beispielsweise bei Consumer-Web-Apps und mobilen Anwendungen, kann GraphQL seine Stärken ausspielen. Die Möglichkeiten, die durch das statisch typisierte Schema entstehen, beispielsweise mächtige Entwickler-Tools, sind vielversprechend. Auch die starke Entkopplung von Komponenten, die durch GraphQL ermöglicht bzw. vereinfacht wird, verspricht einen Mehrwert bezüglich Wiederverwendbarkeit und Wartbarkeit. Es wird spannend sein zu sehen, welche weiteren Entwicklungen und Ideen wir in dieser Richtung in Zukunft erwarten können.

Quellen

[1] *https://developer.github.com/v4*
[2] *https://github.com/sogko/graphql-schema-language-cheat-sheet*
[3] *https://github.com/jcrygier/graphql-jpa*
[4] *https://github.com/graphql/graphiql*
[5] *https://github.com/apollographql/eslint-plugin-graphql*
[6] *https://github.com/creditkarma/graphql-validator*
[7] *https://facebook.github.io/relay*
[8] *http://dev.apollodata.com*

Manuel Mauky
manuel.mauky@saxsys.de

Manuel Mauky arbeitet seit dem Jahr 2010 als Software-Entwickler bei der Saxonia Systems AG in Görlitz. Er beschäftigt sich mit allen Aspekten der Anwendungsentwicklung mit einem Fokus auf das Frontend, zum Beispiel mit JavaFX und JavaScript. Außerdem interessiert sich für funktionale Programmierung und neue Programmiersprachen. Er ist einer der Organisatoren der Görlitzer Java User Group und steuert dafür und auch für andere User Groups und Konferenzen regelmäßig Vorträge bei.