Hochschule für Technik und Wirtschaft Dresden
Fachbereich Informatik/Mathematik

Bachelorarbeit

im Studiengang Wirtschaftsinformatik

Thema: Vergleich der Web API Ansätze REST und GraphQL

eingereicht von: Fabian Meyertöns

eingereicht am: 9. Dezember 2019

Betreuer: Prof. Dr. Thomas Wiedemann

2. Gutachter: Prof. Dr. Jürgen Anke

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Zielstellung

- Entwicklung von Web Anwendungen über die Zeit vom Monolith zu Service-orientierter Architektur
- Entwicklung vom Thin-Client zum Fat-Client, vom Server zu Services.
- Single Page Applikationen gesamte Kommunikation über Web APIs
- Vielzahl von internen Services im Unternehmen und externen Serviceanbietern führt zu wachsender Komplexität
- Einheitliche Kommunikation zwischen Client und Services ist wichtig.
- REST hat sich etabliert als Architekturkonzept, bleibt aber Implementierungsdetails schuldig. Verschiedene Standards und Dateiformate versuchen Einheitlichkeit zu schaffen und Komplexität zu verringern.
- GraphQL, 15 Jahre nach REST veröffentlicht, schafft festes Regelwerk/Protokoll für Client-Server Kommunikation, erfordert aber Umdenken und sehr verschiedene Implementierung.
- Frage für bestehende Anwendungen und APIs nach Migration zu GraphQL.
- Ersetzt GraphQL REST? Bei welchen Anwendungszwecken kann es als Ersatz dienen, bei welchen nicht?
- Welche neuen Probleme entstehen erst durch GraphQL?
- Ist ein gemeinsamer Einsatz von REST und GraphQL sinnvoll und möglich?

Das Internet (Netz, Web) und Web-Anwendungen unterliegen einem starken Wandel. Die Kommerzialisierung, zunehmende Verbreitung und Erweiterung der Möglichkeiten haben zu einer veränderten Nutzung und Entwicklung von Internetanwendungen beigetragen. Vom Netzwerk zum Teilen von Hypertextdokumenten ist es zur Plattform/Medium (TODO) für alle Arten interaktiver betrieblicher und sozialer Software geworden.

Um der zunehmenden Größe, Komplexität und mangelnden Wartbarkeit der Anwendungen Herr zu werden, hat sich auch in der Webentwicklung der anerkannte Grundsatz „Teile und Herrsche“ durchgesetzt. Namentlich ist der Trend von sogenannter monolithischer zu service-orientierter Software zu beobachten. Verbesserungen in der Skriptausführung von Webbrowsern und die Weiterentwicklung von JavaScript selbst hatte außerdem

zur Folge, dass immer mehr Teile der Programme auf dem Client ausgeführt werden. Zeuge dieser Entwicklung sind die vielen JavaScript Frameworks (z.B. Angular, React, Vue), welche besonders innerhalb der letzten 10 Jahre populär geworden sind und mittlerweile zum Standard der Webentwicklung gehören, sowie Projekte wie Electron und NativeScript, welche Webtechnologien nutzen, um native Anwendungen zu entwickeln. (TODO: Stackoverflow survey als Quelle) Mit diesen Frameworks entwickelte Single Page Applications (SPA) generieren Ansichten und Dialoge auf Basis von Daten, welche über Web APIs mit dem Server ausgetauscht werden. Oft kommunizieren Anwendungen mit einer Vielzahl von Webservices, z. B. um unternehmensinterne oder externe Datenquellen oder Dienste zu integrieren.

Um eine einheitliche Kommunikationsschnittstelle für die Interaktion von Clientanwendungen und Webservices zu schaffen, werden APIs häufig nach dem Representational State Transfer Architekturkonzept (REST) entwickelt. Da REST aber nur Rahmenbedingungen und die Grundgedanken für Web-APIs liefert, finden sich in der Praxis viele Formate und Modelle, welche versuchen Implementierungsdetails und Best-Practices auszuführen. Die Bezeichnung REST-API ist somit aufgeweicht worden und Entwickler sind, aufgrund mangelnder Standards, zu API spezifischen Anpassungen gezwungen.

GraphQL wurde 2015 von Facebook veröffentlicht und bringt eine Spezifikation und Abfragesprache für die Client-Server Kommunikation. Die Implementierung einer GraphQL-API unterscheidet sich gravierend von bestehenden REST-APIs, mittlerweile existieren aber wichtige Entwicklerwerkzeuge und die ersten GraphQL-APIs haben sich in der Praxis bewährt. (TODO: Github als Quelle)

Die vorliegende Arbeit soll mehrere Fragen bezüglich der beiden Ansätze für bestehende Web-APIs und Neuentwicklungen klären:

1. Welche Vorteile bietet GraphQL?
2. Welche Probleme entstehen erst durch GraphQL?
3. Für welche Anwendungszwecke kann eine GraphQL-API eine REST-API ersetzen?
4. Ist ein gemeinsamer Einsatz der beiden Ansätze möglich und sinnvoll?

1.2 Aufbau der Arbeit

- Betrachtung der Entwicklung von Web Anwendungen mit der Client-Server Architektur als Grundlage
- Abgrenzung des Begriffes API und Differenzierung von anderen API Ansätzen
- Das REST Architekturkonzept als Grundlage für das Web und APIs
- GraphQL als Alternative, seine Funktionsweise
- Vergleich von REST und GraphQL
- Welchen klassischen Problemen müssen sich API Entwickler stellen?

- Welche Probleme von REST löst GraphQL?
- Welche Vorteile hat REST gegenüber GraphQL?
- Vorstellung einer Auswahl von Tools und Bibliotheken, die verschiedene Probleme von REST und GraphQL lösen bzw. die Entwicklung vereinfachen.
- Untersuchung der Kompatibilität von Bibliotheken
- Tests von REST und GraphQL in verschiedenen Szenarien.
- Kombinierter Einsatz von GraphQL und REST

2 Vorbetrachtungen

2.1 Client-Server Architektur

- Client-Server ist ein verteiltes System
- Zwischen Client und Server geschieht Nachrichtenaustausch
- Client fordert eine Operation vom Server an. Server sendet Resultat der Operation an den Client zurück.
- Client initiiert die Interaktion. Server reagiert.
- Mehrere Clients können den gleichen Server nutzen. Abbildung aus 'Grundkurs verteilte Systeme'!
- Client kann mehrere Server benutzen. Server kann in anderer Interaktion selbst zum Client/Vermittler werden.
- Vorteile
  - getrennte Entwicklung
  - unabhängige Ausfälle
  - Festgelegte Rollenverteilung: Client ist Konsument. Server ist Produzent.
- Herausforderung: einheitliches Kommunikationsprotokoll

Die Basis für die meisten Anwendungen im Web bildet die Client-Server Architektur. Der Nachrichtenaustausch in einem verteilten System wird dabei auf zwei Akteure und eine Interaktion fokussiert. Der Client initiiert die Interaktion, indem er eine Nachricht an den Server sendet. Der Server reagiert, indem er mit dem Resultat der Auswertung der Nachricht antwortet. Ein Client kann mit mehreren Servern in Interaktion treten und mehrere Clients können mit dem gleichen Server kommunizieren. Durch die Anfrage werden die Rollen von Client und Server festgelegt, welche jedoch nur für die Dauer der Interaktion gelten. So kann ein Akteur, der bei einem Nachrichtenaustausch als Server angefragt wird, während eines anderen selbst als Client die Kommunikation beginnen. Bedingung für die Client-Server Architektur ist, dass dem Client zu Beginn der Kommunikation, der Server bekannt ist und dass beide das gleiche Kommunikationsprotokoll unterstützen.

Die Entwicklung einer Anwendung nach dieser Architektur hat ein erklärtes Ziel und Hauptvorteil: Die Trennung der Verantwortlichkeiten.[vgl. 1]

Client und Server sind logisch und oft auch physikalisch getrennte Teile einer Anwendung und können unabhängig voneinander entwickelt werden, solange sie ihre Kommunikationsschnittstelle wahren. Die damit verbundene Trennung der Verantwortlichkeiten führt zu einer Reduzierung der Komplexität der einzelnen Anwendungen. Da Client und Server unabhängig voneinander ausgeführt werden, beeinflusst der Ausfall des einen nicht die Stabilität des anderen. Wird der Client als Benutzeroberfläche und der Server als Datenspeicher genutzt, so können beispielsweise mehrere Clients für verschiedene Geräte oder Anwendungsfälle entwickelt und optimiert werden, wodurch die Portabilität der Oberfläche verbessert wird. Die Verteilung auf mehrere Geräte erhöht außerdem die Skalierbarkeit des Gesamtsystems.

2.2 Web APIs

- API bezeichnet Application Programming Interface
- Grundsatz ist die Kommunikation zweier Programme zur Konsumierung von anderem Quellcode, Abstraktion und Verstecken von Implementierungsdetails und Komplexität.
- Frameworks und Bibliotheken vieler Programmiersprachen bieten oft benötigte Funktionalität. Kommunikation besteht aus Aufruf mit Parametern und Antwort mit Ergebnis.
- Die Arbeit beschäftigt sich nur mit API von verteilten Systemen.
- Hauptaugenmerk auf Systemen mit Fat-Client. Großer Teil der Anwendungslogik auf Clientseite. Server dient als Datenspeicher. Hauptinteraktionen mit CRUD (Create, Read, Update, Delete)
- Popularität von Cloudservices und öffentlichen APIs bzw. Interaktion mit externen Services

Eine Software kann einen Teil seiner Funktionalität über Schnittstellen externen Programmen zur Verfügung stellen. Solche Application Programming Interfaces (API) ermöglichen die Kommunikation zwischen Programmen durch das parametrisierte Aufrufen der öffentlich gemachten Funktionen einerseits und das Antworten mit dem Ergebnis der Funktion andererseits. Das Verstecken der zugrundeliegenden Komplexität ist ein großer Vorteil der Verwendung von APIs und führt zu Unabhängigkeit von Implementierungsdetails. So ist es z.B. die Aufgabe eines Betriebssystems, Programmen den Zugriff auf Hardwarekomponenten, wie Dateisystem, Netzwerk, Sensoren und Peripheriegeräte, zu abstrahieren und bereitzustellen. Externe Programme werden durch die Benutzung jedoch abhängig von der Schnittstelle, welche daher als Kommunikationsobjekt Vertragscharakter bekommt. Für die meisten Programmiersprachen existieren Programmbibliotheken, die zur Entwicklungszeit grundlegende und immerwieder benötigte Funktionalität bereitstellen, aber besonders Cloudservices und öffentliche APIs haben in den letzten

Jahren an Popularität erlangt, da sie mit HTTP eine sprachunabhängige Schnittstelle bieten. Die vorliegende Arbeit beschränkt sich auf API von verteilten Systemen im Web. Web-Frameworks werden häufig auch als Fat-Client Anwendungen bezeichnet, da ein beträchtlicher Teil der Programmlogik, sowie die Benutzeroberfläche auf Clientseite ausgeführt wird. Da der Server in solchen Anwendungen als Zugangspunkt und Speicher für Daten dient, sind die APIs geprägt von der Übertragung der Daten zur Anzeige im Client und der Modifikation der Daten auf dem Server. Gemäß den Hauptinteraktionen werden diese APIs auch als CRUD (Create, Read, Update, Delete) APIs bezeichnet.

2.3 Abgrenzung zu anderen Web API Ansätzen

- SOAP (Simple Object Access Protocol), XML basiert, nutzt nur POST
- RPC (Remote Procedure Call), Zentraler Punkt ist das Aufrufen von Anwendungslogik auf dem Server, nicht Datentransport.

3 Das REST Architekturkonzept

3.1 Entstehung

Das Architekturkonzept des Representational State Transfer wurde erstmals im Jahr 2000 von Roy T. Fielding erwähnt, definiert und mit dem Akronym REST bezeichnet.[1, S. 76] Es enthält eine Reihe von Prinzipien für die Entwicklung von verteilten Systemen im Web mit besonderem Fokus auf den Schnittstellen zwischen den einzelnen Komponenten. Dabei baut REST auf der Client-Server-Architektur auf und hatte durch die Arbeit vib Fielding in den HTTP- und URI-Arbeitsgruppen direkten Einfluss auf die Entwicklung von der beiden Standards, wodurch jene Schnittstellen im Web definiert werden.[vgl. 1, 4, 105f.] REST beschreibt größtenteils die Architektur des Web selbst und ist daher nicht auf APIs ausgelegt oder beschränkt.

3.2 Grundlagen

Ziele von REST

Die Ziele und Vorteile der REST Architektur lassen sich folgendermaßen zusammenfassen:

> REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. [1, S. 105]

Diese Ziele werden erreicht durch eine Reihe von Anforderungen.

Komponenten und Prinzipien

Die REST Architektur baut auf der weit verbreiteten Client-Server Architektur auf und definiert den Begriff der Ressource als Abstraktion für jede Art von Information, die Ziel einer Client-Anfrage ist.[vgl. 1, 88f.] Digitale „Dokumente oder Bilder, [...] eine Sammlung anderer Ressourcen [...][und selbst] nicht-virtuelle Objekte“ [1, S. 88] können solche Ressourcen sein. Eine Ressource wird durch einen Bezeichner bekannt gemacht und ist damit für einen Client abrufbar. Der Bezeichner bleibt gleich, selbst wenn sich die damit identifizierte Ressource ändert. Die Kommunikation einer Ressource vom Server zum Client erfolgt über eine Repräsentation der Ressource, d.h. ein Datenformat, welches vom

Server festgelegt, oder durch Content-Negotiation zwischen Client und Server ausgehandelt wird. Somit bleibt der Ursprung der Daten hinter der Serverschnittstelle versteckt. Die Antwort des Servers enthält die Daten der Repräsentation und deren Metadaten. Anfrage und Antwort enthalten außerdem Kontrolldaten, z. B. HTTP Request Methods, Status Codes und Header, um die Bedeutung der Nachricht zu vermitteln.[vgl. 1, 90f.]

Tabelle 3.1: REST Data Elements

Data Element	**Web Equivalent**	**Example**
resource	the intended conceptual target of a hypertext reference	a list of events
resource identifier	URI	http://example.com/events
representation data	HTML, JSON Document, PNG image	{events:[{title:...},{...}]}
representation metadata	media type, last-modified time, ETag	Content-Type: application/json Content-Encoding: gzip
control data	if-modified-since, cache-control, request method	GET, Cache-Control: no-cache

REST ist zustandslos, sodass jede Anfrage vom Client zum Server alle für deren Verarbeitung notwendigen Informationen beinhalten muss. Ein Client speichert gewöhnlich seinen Zustand, während auf Serverseite keine Zustandsinformationen über mehrere Anfragen hinweg gespeichert werden, wie es zum Beispiel bei Sessions der Fall ist. Aufgrund dieser Eigenschaft wird im Zusammenhang mit REST auch von einer "Client-Cache-Stateless-Server[-Architektur]" [1, S. 83] gesprochen. Hieraus ergeben sich drei Vorteile:[vgl. 1, S. 79]

- Sichtbarkeit: Jede Anfrage kann separat untersucht und gecacht werden, ohne weitere Informationen zu benötigen.
- Zuverlässigkeit: Anfragen können atomar betrachtet und bei Teilfehlern kann ein stabiler Systemzustand leicht wiederhergestellt werden.
- Skalierbarkeit: Da ein Server Zustandsinformationen nur für die Dauer der Verarbeitung einer Anfrage speichert, können Ressourcen schnell wieder freigegeben werden.

Nachteil dieser Anforderung ist eine verringerte Netzwerkperformance, da bei sequenziellen Anfragen an einen Server, Daten zur Clientidentifikation, Authentifizierung oder aus vorangegangene Antworten erneut gesendet werden müssen. Die verringerte Kopplung führt ebenfalls dazu, dass die Kontrolle des Servers über das Verhalten der Clientanwendung reduziert wird.

Clientseitiges Speichern von erhaltenen Daten (Caching) erlaubt die Wiederverwendung früherer Serverantworten für zukünftige, gleiche Anfragen und ist sinnvoll, da die gefühlte Performance durch Verringerung der durchschnittlichen Latenz erhöht wird. Dadurch wird die Effizienz und Skalierbarkeit der Anwendung erhöht, obwohl die Latenz jeder einzelnen Anfrage durch den Cache-Lookup erhöht wird. Je mehr die gespeicherten Daten von den tatsächlichen Daten abweichen, desto mehr verringert sich die Verlässlichkeit derselben.[vgl. 1, S. 80] GET-Requests werden von Webbrowsern implizit gecacht, wobei Server das gewünschte Caching-Verhalten explizit erlauben und verbieten können.

REST beschreibt drei Typen von Komponenten:[vgl. 1, S. 96]

- User Agent: Dies kann ein Webbrowser bzw. die Benutzeranwendung sein, in jedem Fall jedoch der letztendliche Empfänger der Antwort.

- Origin Server: Die endgültige Quelle der Repräsentation einer Ressource und der letztendliche Empfänger von Request, die zu Modifikationen der Daten führen.

- Intermediary (Zwischenkomponenten): Diese befinden sich zwischen User Agent und Origin Server und können sowohl als Client, als auch als Server agieren. Sie leiten Anfragen und Antworten weiter, können diese aber auch modifizieren. Je nach Anwendungsfall spricht man hier von einem Gateway oder Proxy.

Für die Schnittstellen der Komponenten gelten folgende Grundsätze:[vgl. 1, S. 82]

- Ressourcen müssen durch einen Bezeichner eindeutig identifizierbar sein.

- Die Interaktion mit einer Ressource, d.h. das Abfragen und Manipulieren derselben erfolgt über eine Repräsentation der Ressource.

- Nachrichten müssen so selbsterklärend sein, dass sie von der Schnittstelle verstanden und verarbeitet werden können.

- "hypermedia as the engine of application state" [1, S. 82] (HATEOAS): Der Hypertext beschreibt den Zustand der Anwendung und die Möglichkeiten der Veränderung desselben.

Durch die generischen Client- und Serverschnittstellen, die selbstbeschreibenden Nachrichten und die zustandslose Kommunikation kann eine REST Anwendung leicht von Schichten profitieren. Keine einzelne Komponente benötigt Kenntnis des gesamten Informationsweges, sondern sieht nur die Komponenten, mit denen sie direkt interagiert, sodass zwischen User-Agent und Origin-Server Zwischenkomponenten eingeführt werden können, um spezifische Aufgaben, wie Caching, Kapselung, Load Balancing und Datentransformation, zu übernehmen.[vgl. 1, S. 99] Die so verbesserte Skalierbarkeit des Servers bringt jedoch mehr Datenverarbeitung durch die Zwischenkomponenten und somit höhere Latenzzeiten mit sich.[vgl. 1, S. 83, 98]

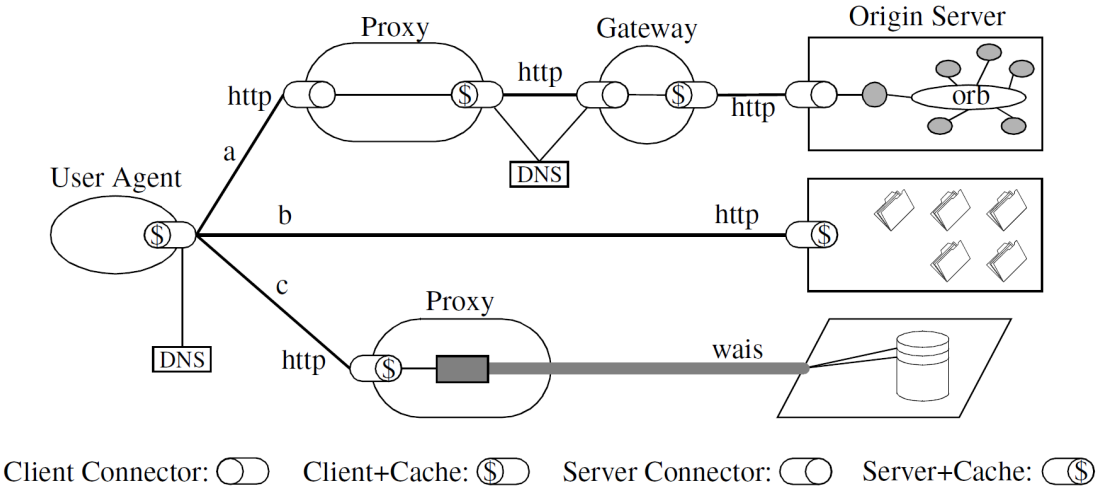

Abbildung 3.1: REST [1, S. 84]

3.3 Implementierung von RESTful APIs

Die REST Architektur ist generell protokollunabhängig, durch die Arbeit von Fielding an HTTP und URI sind diese jedoch sehr geeignet für die Implementierung und waren auch das erste Anwendungsfeld.[vgl. 1, S. 109, 116] Eine API, die dem REST Archtekturstil folgt, wird RESTful genannt. Da die Dissertation von Fielding jedoch einige Implementierungsdetails schuldig bleibt bzw. bewusst „Details der Komponentenimplementierung und Protokollsyntax [ignoriert]“ [1, S. 86], wurde der Begriff RESTful API mit der Zeit aufgeweicht und für APIs verwendet, die grundlegende Anforderungen nicht erfüllen.[vgl. 2]

- Richardson Maturity Model ermöglicht Bestimmung wie REST konform Web service (API) ist
- Level 0
- Level 1 URI
- Level 2
  - HTTP Methoden genutzt als Kontrolldaten um Intention auszudrücken
  - CRUD Operationen werden abgedeckt
  - GET: Anfragen einer Repräsentation der Ressource
  - POST: kann zum Erstellen, Modifizieren und Löschen von Ressourcen verwendet werden, schlecht definiert; Funktionsweise in folgende Methoden aufgeteilt
  - PUT: Erstellen/Ersetzen einer Repräsentation
  - PATCH: Modifizieren einer Repräsentation

  - DELETE: Löschen der Ressource
  - GET, PUT, DELETE sind idempotent (gleiches Ergebnis bei mehrmaliger Ausführung). GET ist safe (kein Verändern der Ressource).

- Level 3
  - Hypermedia ermöglicht Navigation durch die API. Client ändert seinen Zustand, indem er URIs (Links) folgt (HATEOAS)
  - keine externe Dokumentation nötig. Links zwischen Dokumenten dokumentieren die Ressourcen
  - Datenformat ist entscheidend. Bestimmte Formate haben native Unterstützung für Links und Forms (HTML, ATOM)
  - Media Type bestimmt Auswertung (und Anzeige) der Antwort. JSON, XML können genutzt werden. Client benötigt Informationen über Datenstruktur, um Links in diesen Dokumenten auszuwerten.

3.4 Verbreitung und Standardisierung

- REST bestimmt nicht welches Format benutzt werden muss.
- Kein REST Standard
- REST APIs nutzen Web Standards (HTTP, URI, Hypermedia)
- XML und HTML zur direkten Anzeige geeignet. JSON beliebter geworden, das leichter für Menschen und Maschinen zu lesen
- verschiedene Ansätze um Struktur von JSON Dokumenten zur Verwendung in APIs zu definieren. Teilweise miteinander verwendbar (definieren verschiedene Aspekte der Kommunikation)
- OpenAPI
- JSON:API
- Abbildung Beispiel Request und Response

REST bleibt Implementierungsdetails schuldig[vgl. 1, S. 86]

4 GraphQL

4.1 Entwicklung und Grundgedanke

- GraphQL ist Abfragesprache für APIs und Laufzeitumgebung um auf Abfragen zu antworten
- Server stellt Schema = komplette Beschreibung der Datenstruktur
- Client sendet beliebige Abfrage und erhält exakt die angefragten Daten
- Eine Anfrage für alle nötigen Daten (für eine View, UI basiert); Vorteil bei langsamen mobilen Netzwerken
- 2012 von Facebook entwickelt und eingesetzt
- 2015 open source, GraphQL Foundation, Spec auf Github weiterentwickelt, Juni 2018 letzter Release
- Besteht aus Typsystem, Abfragesprache, Ausführungssemantik, statischer Validierung und Typintrospektion

4.2 Spezifikation und Funktionsweise

- Typsystem
  - Typsystem und GraphQL Schema drücken aus, welche Objekte die API zu Verfügung stellt
  - Schema besteht aus 'type', 'enum' und 'interface'. 'type' kann 'interface' implementieren
  - jeder Type (und Interface) ist Ansammlung von Feldern
  - 'null' ist erlaubter Wert. Non-nullable Feld wird mit '!' markiert
  - Einstiegspunkt (Top level) in Typsystem ist Objekttyp, Name nach Konvention 'query'
  - Felder auf query Typ sind mögliche Operationen; Argumente möglich
- Query Syntax
  - Abbildung Beispiel Query
  - GraphQL Abfrage beschreibt deklarativ welche Daten erwartet werden

  - Antwort ist JSON mit der gleichen Struktur
  - Abfragen können geschachtelt werden
  - Fragmente
    * Abbildung Beispiel Fragment
    * verhindert Dopplung von mehreren Feldern in Abfrage
    * ermöglicht typbasierte Feldselektion

- Introspektion
  - Spezialfelder beginnend mit doppelt Unterstrich
  - __schema, __typename
  - Metadaten über GraphQL Schemas
  - Sinn ist Nutzung durch Entwicklungstools
  - ermöglicht statische Validierung: GraphQL Abfrage kann zu Entwicklungszeit geprüft werden
- Referenzen werden unsichtbar, da von GraphQL Server automatisch aufgelöst (Performance beachten!)
- ein Endpunkt (kein Nutzen von URIs)
- jede Abfrage mit POST (kein Nutzen von HTTP Methoden)
- Semantik der Abfrage von Server ausgewertet (welche der CRUD Operationen)

4.3 Server-Execution

5 Vergleich

5.1 Testumgebung

Für den Vergleich der beiden API Ansätze wurde eine Single-Page Anwendung mit dem Vue.js Framework als Client und jeweils eine REST API und GraphQL API mit der Node.js Serverlaufzeitumgebung entwickelt. Als Datenquelle dient zum einen eine MongoDB Datenbank, sowie Dateien im JSON Format. Durch die Verteilung der jeweiligen Softwarekomponenten auf physikalisch unterschiedliche Rechner, lassen sich mehrere Anwendungsfälle abbilden. So wurden die Clientanwendung, die Serveranwendungen und die Datenbank in verschiedenen Kombinationen lokal und von Cloudumgebungen (Azure, Heroku, MongoDB Atlas) gehostet und ausgeführt. Die JSON Dateien als Alternative zur Datenbank ermöglichen es das Kontingent der Clouddatenbank nicht unnötig zu belasten und den durch Datenbankabfragen entstehenden Overhead auf die deutlich schnellere Dateilesezeit zu minimieren. Bei den Messungen wurde auf das Angleichen der Testbedingungen geachtet, besonders die im Zusammenhang mit Clouddiensten gemessenen Werte sind jedoch stark von der Tageszeit und der unbekannten Belastung des Dienstes abhängig.

5.2 Abfrageflexibilität

- nicht nur Abfrage (GET) sondern auch Änderungsoperationen (POST,...)
- GraphQL
  - keine Wildcards (alle Felder eines Objekts)
  - als Abfragesprache gedacht
  - unterscheidet Feldselektion, Sortieren, Filtern, Optionen
  - keine Syntax für Sortieren/Filtern, aber Möglichkeit über Feldargumente
  - Bsp: height (unit:FOOT)
  - sehr anpassbar für verschiedene Apps bzw. öffentliche API (Anforderungen unbekannt)
  - include ist quasi Pflicht, effizient und konsistent
  - an Query sind Performanceprobleme evtl. schwer erkennbar (siehe Tools)
- REST
  - oft einfacher Anfang. Mit steigender Komplexität werden Abfragesprachen typische Konstrukte eingebaut

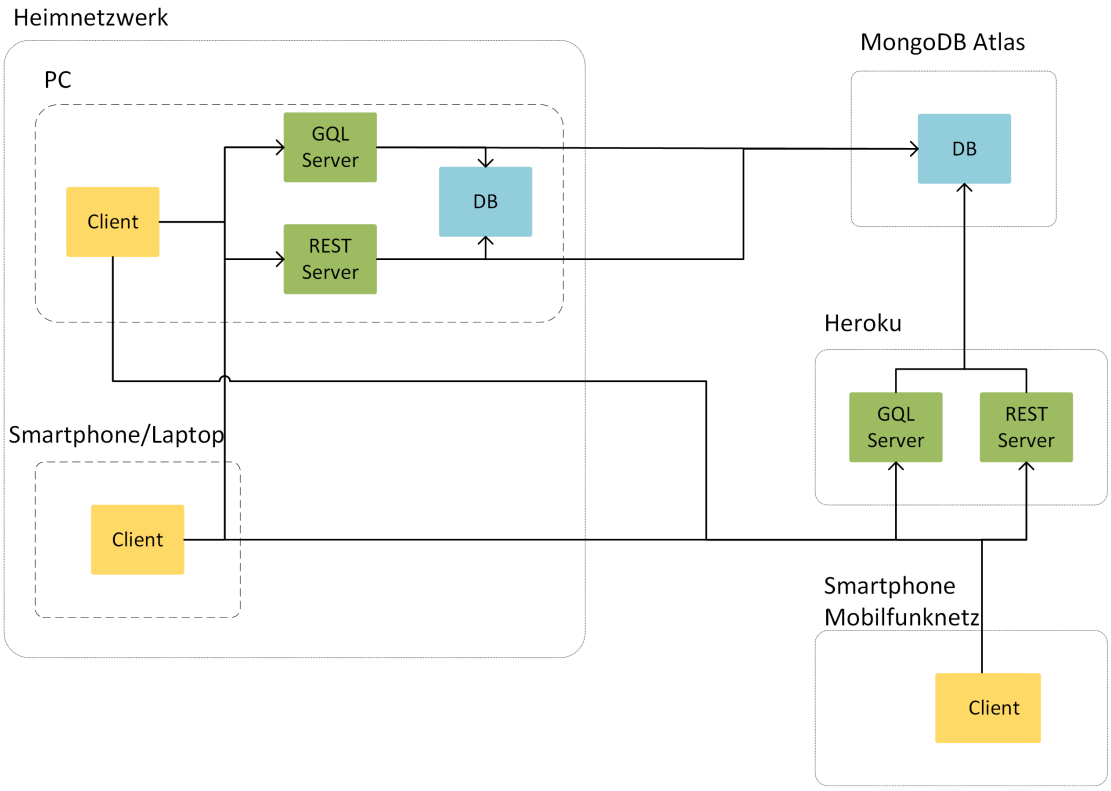

Abbildung 5.1: Testumgebung

  - fields=“ ” für Feldselektion
  - field=“ ” zum Filtern
  - sort=[field],sort-dir=desc zum Sortieren
  - option=“ ” für Optionen
  - includes empfohlen bei JSON:API, aufbrechen von HATEOAS?
  - includes oder extra Endpunkt ist Entwurfsentscheidung
  - Anforderungen für Client App können spezifisch werden, extra API pro Client
- Upload ist schwierig für GraphQL (serialization), REST kann multipart/form-data header nutzen

Bei der Entwicklung von Client-Anwendungen ist die Flexibilität der genutzten API ein entscheidender Einflussfaktor. Dabei steht besonders die Flexibilität für das Generieren von Abfragen im Vordergrund, aber auch von Operationen zum Modifizieren von Ressourcen.

GraphQL wurde als Abfragesprache entwickelt. Eines der Entwicklungsziele ist es, dass eine Clientanwendung mit einer einzigen Abfrage alle für eine Ansicht nötigen Daten erhalten kann. Der Server gewährt Zugriff auf das Datenschema, welches alle verfügbaren

Daten und deren Relationen darstellt, sowie die Einstiegspunkte (Root Query), auf deren Basis ein Client seine Anfrage aufbauen kann. Das Schema stellt den Rahmen für alle Abfragen dar. Alle Daten, die der Server zur Verfügung stellt, werden durch Typen und deren Felder repräsentiert. Jedes Feld, welches eine Clientanwendung erhalten möchte, muss in der Query angegeben werden. Es existiert keine Möglichkeit alle Felder eines Typs zu erhalten ohne jedes davon anzugeben. Der Client ist damit eng an das Schema gekoppelt, erhält aber auch nur die Daten, die tatsächlich angefragt wurden. Die Möglichkeit eines GraphQL-Servers beliebige Abfragen innerhalb des Schemas auszuwerten eignet sich besonders für öffentliche APIs, bei denen dem API-Entwickler die individuellen Anforderungen und damit die Abfragen verschiedenster Anwendungen, im Voraus nicht bekannt sind. Die Relationen zwischen den Datentypen werden durch das Schema festgelegt. Eine Anfrage kann jedoch beliebig viele, unabhängige Queries beinhalten, z. B. um verschiedene Daten zu erhalten, die in keiner Beziehung zueinander stehen oder deren Beziehung durch das Schema nicht dargestellt wird. Der Server kann für jedes Feld Parameter anbieten, welche die Auswertung des Feldwertes verändern. Über solche Feldargumente können Optionen, die Geschäftslogik darstellen, aber auch allgemeine Abfragefunktionen, wie z. B. Sortieren und Filtern zur Verfügung gestellt werden.

Während das Anfragen von Daten sehr viel Flexibilität für Clients bietet, ist das Modifizieren von Ressourcen nach Konvention nur über einzelne Mutations möglich. Üblicherweise wird für jede Operation (Erstellen, Verändern und Löschen) einer Ressource eine separate Mutation zur Verfügung gestellt, welche durch den Namen eindeutig identifiziert wird. Die Antwort auf eine Mutation kann wieder ein Objekt sein und genutzt werden um das Resultat der Operation zu übermitteln oder den neuen Stand der Ressource abzufragen.

In einer REST API ist jede Ressource über einen separaten Endpunkt erreichbar. Die Antwort auf einen GET Request an einen Endpunkt enthält alle Felder der Ressource bzw. die komplette Repräsentation. In Beziehung stehende Daten, die nicht Teil einer Ressource sind, müssen über mehrere Requests vom Server angefragt werden. Der Grundgedanke von HATEOAS ist, dass diese verbundenen Ressourcen Verweise aufeinander enthalten, wodurch ihre Beziehung dargestellt wird. Das Zusammenbauen von Requests anhand der Links in erhaltenen Ressourcen ermöglicht große Flexibilität, erfordert jedoch dass Daten erst ausgewertet werden müssen und danach erst neue Anfragen abgesetzt werden können. Dadurch ist das Datenmodell nicht durch ein Schema vorgegeben, sondern kann zur Laufzeit geändert und explorativ genutzt werden, da es selbstdokumentierend ist. Oftmals ist bei der Entwicklung von Clientanwendungen die erforderliche Datenstruktur jedoch bekannt und ist es wünschenswert möglichst alle benötigten Informationen von wenigen Endpunkten zu erhalten, um die Performance zu steigern. Dazu gibt es verschiedene Ansätze, wie der Query-String einer URI genutzt werden kann, um in Beziehung stehende Daten mit einem einzigen Request von einem Endpunkt zu erhalten. Weder REST noch HTTP spezifizieren Syntax und Semantik des Query Strings bzw. betrachten diese als Implementierungsdetail. Query-Parameter und Matrixparameter werden gebraucht, um die Möglichkeiten von Abfragesprachen für REST zu implementieren. Sie ermöglichen z. B. die gewünschten Felder des angefragten Objektes, ein Feld nach dem sortiert, oder ein Feld und Wert nach dem gefiltert wer-

den soll, anzugeben. Über sogenannte „includes“ lassen sich bei JSON:API verbundene Objekte in die Antwort einbeziehen. Includes oder sind außerdem eine effektive Lösung für das N+1 Problem. Diese speziellen Auswertungen des Query-Strings müssen durch den REST Server angeboten werden und sind aus Entwurfssicht ein Tausch von erhöhter Abfragekomplexität und Clientflexibilität gegen Serverperformance. Ein Endpunkt kann alle Methoden zum Abfragen und Ändern einer Ressource bereitstellen und durch die HTTP Verben semantisch unterscheiden. Das Veröffentlichen einer neuen Ressource geschieht durch das Einführen eines neuen Endpunktes.

5.3 Weiterentwicklung und Versionierung

- GraphQL
  - neue Felder hinzufügen ohne existierende Queries zu beeinflussen
  - neue Felder nicht automatisch gesendet
  - Felder als deprecated markieren → Tool kann Entwickler warnen
  - Monitoring auf Feldlevel möglich
- REST
  - Monitoring: werden sparse fieldsets oder includes verwendet können genutzte Felder aufgezeichnet werden
  - neue Version, neuer Endpunkt example.com/v2/contacts/...
  - für kleine Veränderungen ungeeignet
  - Hinzufügen von Ressourcen → keine Versionierung notwendig
  - Semantische Versionierung
  - Postelsches Gesetz
  - Hinzufügen/Umordnen von einzelnen Elementen sollte Client akzeptieren
  - Versionierung über Accept-Header, URI oder query string

Langlebige Software befindet sich in ständiger Entwicklung. Ein Server muss auf Veränderungen des der API zugrunde liegenden Datenmodells reagieren können und diese gegebenenfalls Clients zugänglich machen. Neue Anforderungen an die API erfordern das Erweitern der Abfragemöglichkeiten, oder das Verringern derselben, wenn alte Funktionalität nicht mehr unterstützt oder durch neue ersetzt werden soll. Da jede API einen Vertrag zwischen Server und Client darstellt, sollten Veränderungen einem festgelegten Ablauf und Kommunikationskanal folgen, um sukzessive Migration zu gewährleisten und Abfragefehler zu vermeiden. Zu diesem Zweck wird in der Softwareentwicklung häufig Versionierung verwendet, bei der durch eine Zahl oder Zahlenkombination ein bestimmter Stand der Software bereitgestellt oder abgefragt wird.

Um eine REST API um zusätzliche Ressourcen zu erweitern, ist keine Versionierung notwendig. Stattdessen wird ein neuer Endpunkt hinzugefügt und über die Verlinkung

von anderen Ressourcen, oder als Einstiegspunkt in der Dokumentation bekanntgemacht. Eine API Clientanwendung sollte so entwickelt werden, dass eine Antwort ausgewertet werden kann, solange sie semantisch eindeutig verständlich ist (Postel'sches Gesetz). Das Umordnen und Hinzufügen von Elementen innerhalb einer Antwort sollte somit nicht zu Fehlern führen und die Validierung demgegenüber tolerant sein. Normalerweise muss erst versioniert werden, wenn keine Rückwärtskompatibilität mehr gewährleistet werden kann. Eine neue Version einer Ressource kann über mehrere Wege bereitgestellt und angefragt werden.

1. Jeder Request kann einen Accept-Header enthalten, der zur Identifikation der zu sendenden Repräsentation genutzt werden kann. Sendet ein Client z. B. einen Accept-Header mit dem Wert application/vnd.format.v2+json, so kann der Server durch Content-Negotiation anhand des „v2" die zu sendende Version ermitteln.

2. Die URI als Identifikationsmittel kann ebenfalls genutzt werden, um verschiedene Versionen einer Ressource kenntlich zu machen. Die Anfrage an example.com/v2/events entspricht dem Erstellen eines neuen Endpunktes. Der Server muss nun auf einem weiteren Endpunkt antworten können, die Komplexität jedes einzelnen Endpunktes wird jedoch nicht erhöht.

3. Für kleine Änderungen, wie das Ändern oder Entfernen eines einzelnen Feldes, ist normalerweise kein neuer Endpunkt nötig. Würde für jede dieser verhältnismäßig kleinen Änderungen ein neuer Endpunkt erstellt werden, ginge schnell die Übersichtlichkeit über die Menge der Endpunkte und deren Unterschiede verloren. Die URI der Ressource kann für verschiedene Versionen genutzt werden und stattdessen die Versionsinformationen im Query String transportiert werden (z. B. example.com/events?version=2). Diese Variante teilt den gleichen Nachteil, den alle Query String Optionen haben, dass der Serverhandler für diesen Endpunkt komplexer und die Performance verringert wird. Es ist denkbar, dass nicht nur jede einzelne Option geprüft werden muss, sondern dass die Option für die Version der Ressource auch Auswirkung auf andere Optionen hat, sodass der Server einen Entscheidungsbaum auswerten muss, bevor eine Antwort gesendet werden kann.

Wurde eine neue Version einer Ressource zur Verfügung gestellt, muss diese auch erreichbar sein. Da die Änderungen von Variante 2 und 3 sich allein auf die URI auswirken, müssen in verknüpften Ressourcen die Links zur neuen Version aktualisiert werden. Da Header in einem Link nicht repräsentiert werden können, ist es bei der ersten Variante notwendig, dass der Client über die neue Version informiert wird und allen Anfragen an die Ressource den Accept-Header hinzufügt.

Beim Entfernen von Funktionalität ist es sinnvoll die aktuelle Nutzung derselben durch Monitoring festzustellen und notwendig die geplanten Änderungen zu kommunizieren. Da normalerweise die gesamte Repräsentation einer Ressource als Antwort gesendet wird, kann nur die Nutzung des Endpunktes und nicht die tatsächliche Notwendigkeit einzelner Teile der Ressource bzw. Felder ermittelt werden. Das ist jedoch möglich, wenn bei jeder Anfrage mittels Includes die gewünschten Felder angegeben werden müssen.

Sollen einzelne Elemente einer Ressource nicht mehr gesendet werden, so können diese zeitweise den für den Datentyp üblichen Nullwert enthalten, bevor sie endgültig entfernt werden. Wird ein Endpunkt der API entfernt, so sollte der Server mit dem entsprechenden HTTP Status Code (410 Gone) antworten und alle Links in verknüpften Ressourcen müssen gelöscht werden. Ist eine Ressource lediglich unter einem anderen Endpunkt erreichbar, so kann der Code 310 „Moved Permanently“ den Client automatisch zur neuen URI weiterleiten.

Erklärtes Ziel von GraphQL ist es Versionierung zu verhindern und stattdessen die API kontinuierlich, abwärtskompatibel weiterzuentwickeln. Da jedes Feld, welches ein Client erhalten möchte, in der Query explizit angegeben werden muss, ist detailreiches Monitoring bis auf Feldebene möglich. Im Datenmodell neu eingeführte Felder beeinflussen bestehende Abfragen nicht, können aber jederzeit in diese aufgenommen werden. Felder, die zukünftig aus der API entfernt werden sollen, können mit isDeprecated als veraltet gekennzeichnet und ein Grund als Metainformation angegeben werden. Diese Kennzeichnung beeinflusst die Abfrage des Feldes nicht, kann aber per Introspektion zur Entwicklungszeit von Tools abgefragt und dem Entwickler als Warnung angezeigt werden. Enthält eine Abfrage Felder, die im Schema nicht mehr enthalten sind, schlägt die Abfrage fehl. Änderungen in der Auswertungslogik einzelner Felder können über optionale Feldargumente bereitgestellt werden. Auch diese beeinflussen bestehende Abfragen ohne diese Argumente nicht. Für Feldargumente und Input Typen existiert zur Zeit noch keine Möglichkeit diese als veraltet zu kennzeichnen, ist aber für den nächsten Release der GraphQL Spezifikation geplant. Ein GraphQL Server stellt ein Schema und damit alle darin enthaltenen Abfragen unter einer URI bereit. Ergibt sich die Notwendigkeit eine neue Version unter einer anderen URI zu veröffentlichen, so können Abfragen die Schemata nicht verbinden und werden vom Server nur im Kontext eines Schemas ausgewertet.

5.4 Performance

Zum Vergleich der Abfrageperformance von GraphQL und REST wurden jeweils nacheinander eine Reihe von Abfragen durchgeführt und gemessen, um die Charakteristik der Daten zu erforschen. Die kostenfreie Laufzeitumgebung von Heroku (Dyno) unterliegt einem monatlichen Stundenlimit und wird nach 30 Minuten, in denen keine Anfragen gestellt werden, in den Schlafzustand versetzt. Erhält eine schlafende Dyno eine Anfrage, so wird sie nach einer kurzen Verzögerung wieder aktiv. TODO Zitat Heroku Diese anfängliche Verzögerungszeit ist nach der angegebenen 30 minütigen Pause, in geringerem Maße aber auch nach nur wenigen Sekunden Inaktivität zu beobachten.5.2 Diese anfänglichen Ausreißer sind unabhängig von der verwendeten API, wurden aber aufgrund der hohen Schwankungen in den weiteren Untersuchungen nicht einbezogen. Je nach für die Abfrage verwendetem Gerät unterscheiden sich die Abfragezeiten deutlich.

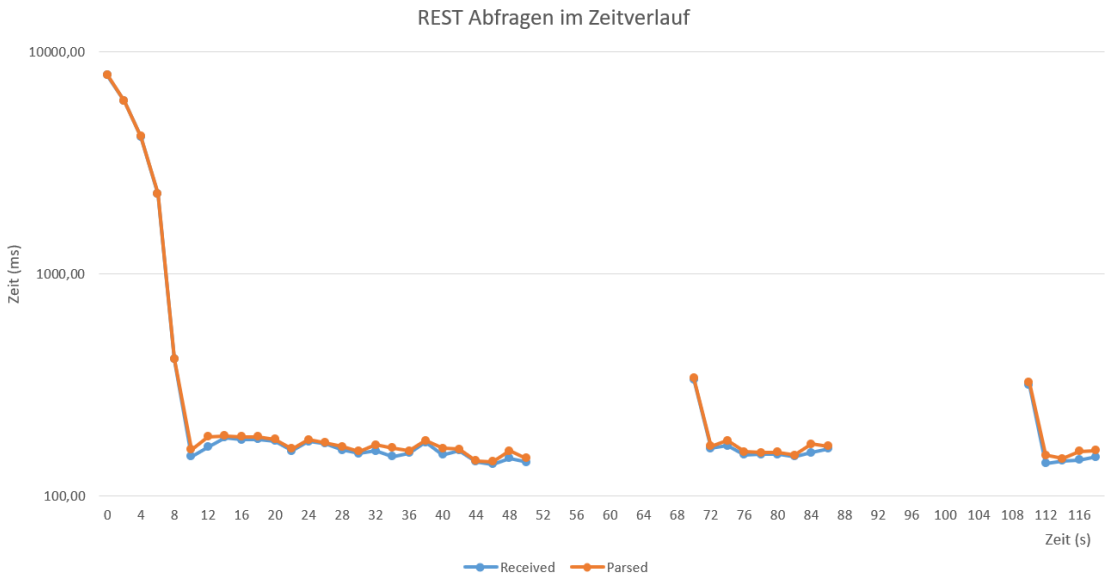

Abbildung 5.2: Abfragen im Zeitverlauf

5.5 Datenaufkommen/Netzwerklast

- Transfer, Verarbeiten und Speichern unnötiger Daten (Felder) sollte vermieden werden
- GraphQL
  - automatisch kleinstmöglicher Request
  - Query muss an Server gesendet werden
- REST
  - standardmäßig gesamte Repräsentation
  - viele APIs bieten Feldselektion an (Partials)
  - query string enthält Feldselektion nach bestimmter Syntax
  - je komplexer, desto mehr Daten
  - jede Ressource ist extra Endpunkt
  - für mehrere Ressourcen müssen mehrere Requests gemacht werden, n+1 Problem
  - includes beziehen verbundene Daten in Response ein → ein Request für mehrere Ressourcen
- GraphQL und REST Partials unterscheiden Objekt und Array nicht → Wissen über API notwendig um Performance einzuschätzen

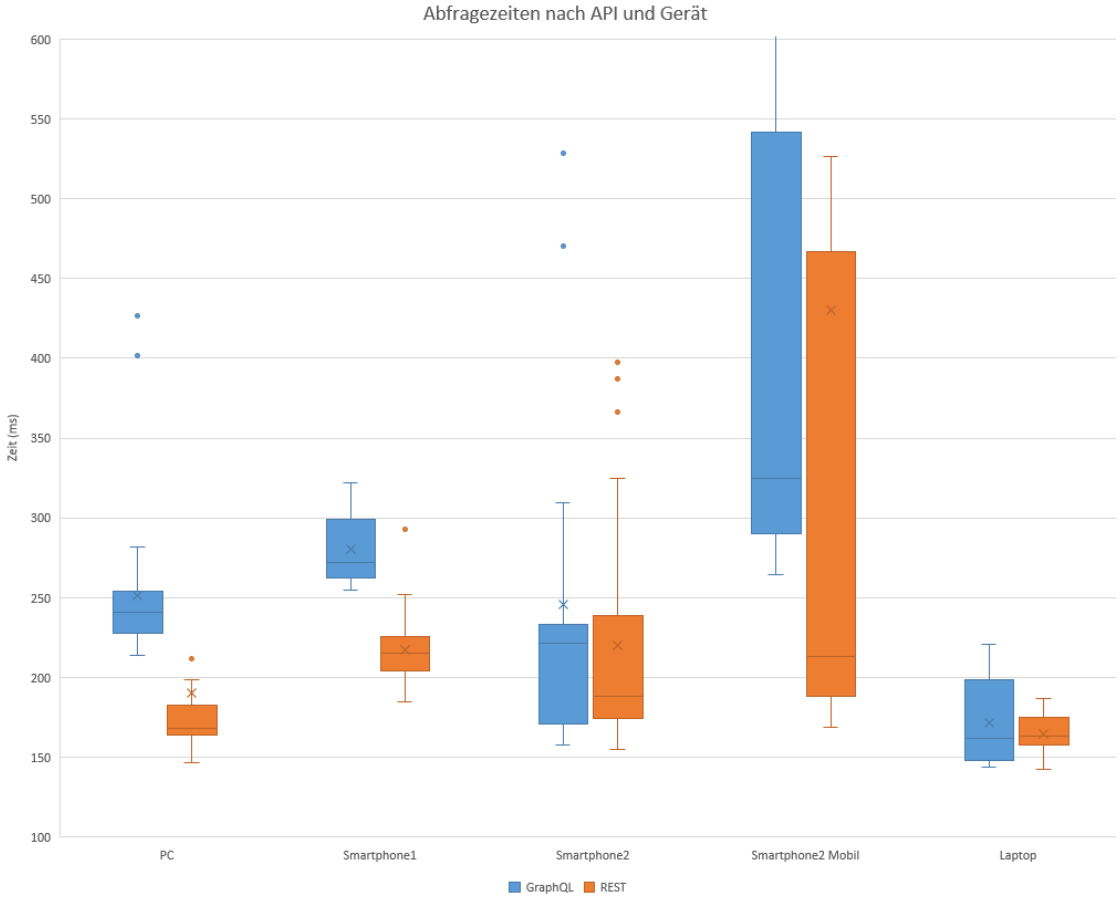

Abbildung 5.3: Abfragen nach Gerät

- mehr Daten um Request genauer zu machen, sinnvoll um deutlich weniger Daten als Response zu erhalten

5.6 Caching

- Flexibilität gegen Caching: je spezieller die Abfrage, desto schwieriger (weniger sinnvoll) caching
- nicht Antwort direkt cachen (HTTP), sondern manuell Objekte anhand ID cachen (JavaScript)
- REST
  - Browser HTTP caching automatisch genutzt
  - Caching basierend auf Endpunkt
  - URL ist cache ID für die Ressource

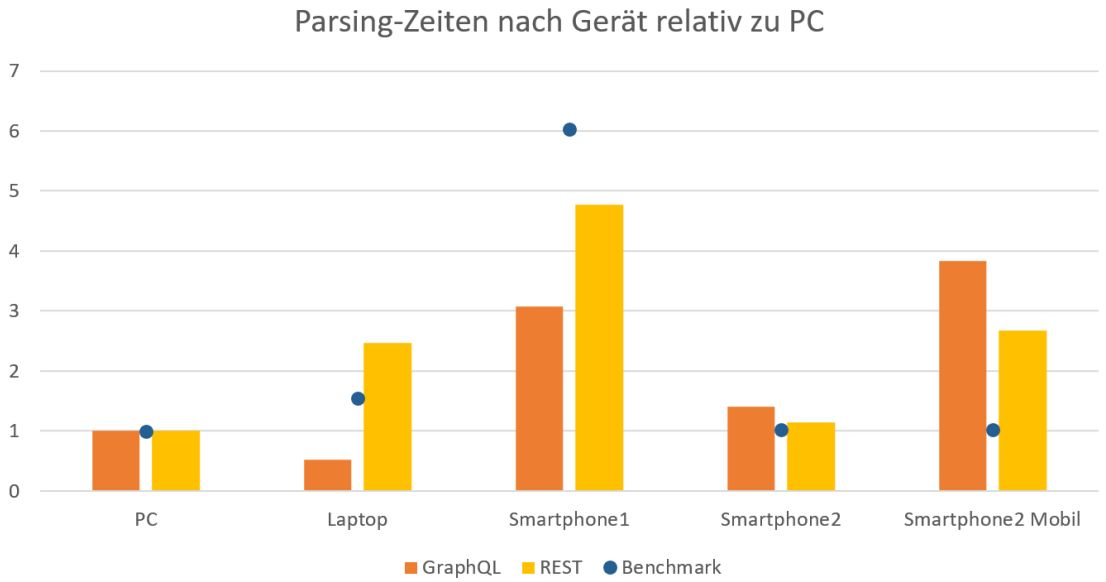

Abbildung 5.4: Testumgebung

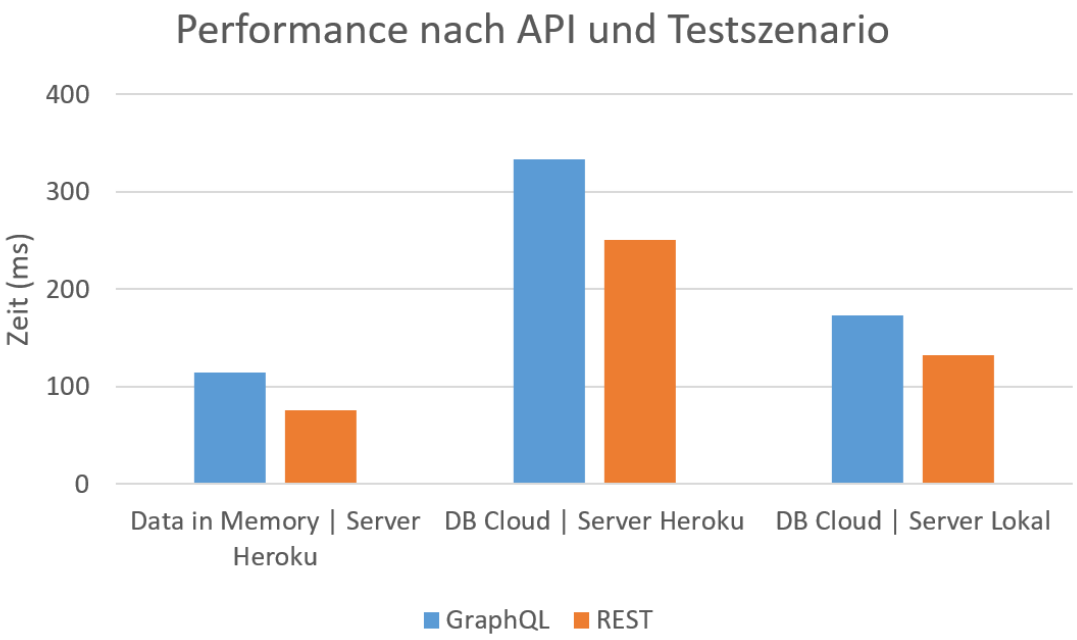

Abbildung 5.5: Testumgebung

- ermöglicht HTTP cache proxies
- je spezieller der query string, desto weniger cache Treffer

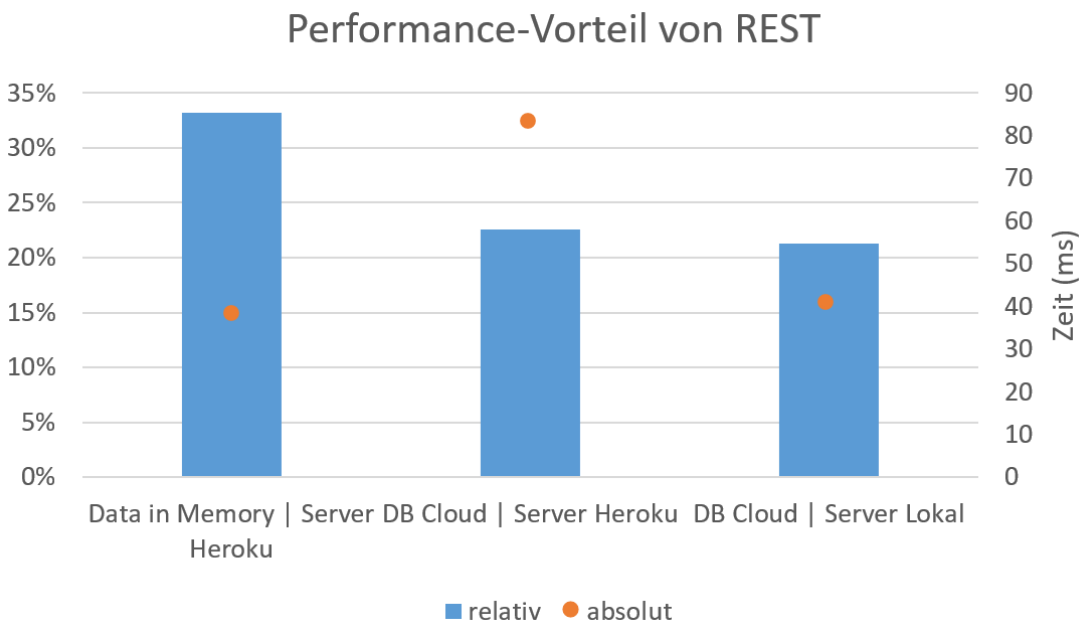

Abbildung 5.6: Testumgebung

- GraphQL
  - POST response wird normalerweise nicht gecacht
  - standardmäßig keine ID für caching vorhanden. Empfehlung: API sollte ID pro Objekt bereitstellen

5.7 Batching/Deduping

- GraphQL queries können parallel ausgewertet werden, Mutations nicht

5.8 Fehlerbehandlung

5.9 Sicherheit

5.10 Kosten

5.11 Lernkurve, Fehlersuche, Community

5.12 Bibliotheken und Tools

- GraphQL

- offizielle Spezifikation vorhanden
- Referenzimplementierung in JavaScript
- Zusatztools von Facebook (Dataloader, Relay)
- Tool kann Abfragekomplexität zu Entwicklungszeit ermitteln (mit vergangenen Messwerten)

6 Fazit und Auswertung

6.1 Zusammenfassung

6.2 Kombinierte Verwendung von GraphQL und REST

6.3 Ausblick

Literaturverzeichnis

[1] Roy Thomas Fielding, „Architectural Styles and the Design of Network-based Software Architectures“, Doctoral dissertation, University of California, Irvine, 2000. Adresse: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (besucht am 25. 10. 2019).

[2] ——, *REST APIs must be hypertext-driven*, 2008. Adresse: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 25. 11. 2019).