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: 10. Oktober 2019

Betreuer: Prof. Dr.-Ing. Thomas Wiedemann

Inhaltsverzeichnis

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?

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

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

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

- Bekanntmachung Roy T. Fielding in Dissertation 2000
- Akronym für Representational State Transfer
- Prinzipien für die Entwicklung von verteilten Systemen. Baut auf bekannten Architekturen auf (Client-Server)

3.2 Grundlagen

- Ziele von REST
  - skalierbare Komponenteninteraktionen
  - generische Interfaces
  - unabhängige Entwicklung der Komponenten
  - Zwischenkomponenten können spezielle Aufgaben übernehmen (Cache, Sicherheit, Schnittstelle zu Altsystemen)
- REST hatte Einfluss (und macht Gebrauch von) HTTP und URI (IRI)
- REST ist zustandslos (daher auch client-stateless-server). Jeder Request von Client zu Server beinhaltet alle notwendigen Informationen, um den Request zu verstehen.
- Client speichert gewöhnlich Zustand. Server behält keine Clientsession bei.
- Vorteile
  - Sichtbarkeit: Requests können einzeln untersucht werden
  - Zuverlässigkeit: einfachere Wiederherstellung bei teilweisen Fehlern
  - Skalierbarkeit: Server kann schnell Ressourcen wieder freigeben. Speichert keine Zustände
- Nachteile
  - verringerte Netzwerkperformance, da mit jedem Request Daten wiederholt
  - verringerte Kontrolle des Servers über Verhalten der Clientanwendung
- Clientseitiges Caching

  - Ressourcen implizit oder explizit gecacht
  - erlaubt Wiederverwendung früherer Serverantworten für zukünftige, gleiche Requests
  - bessere Effizienz und Skalierbarkeit, erhöhte gefühlte Performance durch Verringerung der durchschnittlichen Latenz (jede einzelne Latenz durch Cache-Lookup erhöht)
  - verringerte Verlässlichkeit je stärker gecachte Daten von tatsächlichen Daten abweichen

- 4 Grundsätze für Komponentenschnittstellen
  - Identifizierung von Ressourcen
  - Manipulation von Ressourcen durch ihre Repräsentation
  - Selbsterklärende Nachrichten
  - 'hypermedia as the engine of application state'

- Die Daten werden zum Ort der Verarbeitung geschickt, nicht die Anweisungen zu den Daten.

- Komponenten in REST Architektur sehen nur Komponenten, mit denen sie direkt interagieren (Schichten). Begrenztes Wissen verringert Komplexität. Schichten ermöglichen Kapselung. Zwischenkomponenten (Proxies) können Daten transformieren (wie Pipes/Filter)

- Schichtenarchitektur bedeutet mehr Datenverarbeitung und Latenz

- Client kann Repräsentation der Daten wählen. Ursprung der Daten hinter Serverinterface versteckt

- Ressourcen ist Abstraktion für jede Art Information (Dokumente, Bilder, Sammlung anderer Ressourcen)

- Jede Webseite ist Ressource

- Ressource wird durch Identifier (Bezeichner) bekannt gemacht und abrufbar. Bezeichner ändert sich nicht, wenn sich die Ressource ändert. Mehrere Ressourcen können die gleichen Informationen beinhalten.

- Repräsentation einer Ressource (Antwort des Servers) besteht aus Daten und Metadaten

- Kontrolldaten übermitteln den Zweck der Nachricht oder zum Umgang mit der Nachricht (HTTP Methoden, Status codes, Header)

- REST kennt drei Komponententypen:

  - user agent: Web Browser, Benutzeranwendung, letztendlicher Empfänger der Antwort
  - origin server: endgültige Quelle der Repräsentation der Ressource, letztendlicher Empfänger von Requests, die Modifikationen vornehmen; bietet Schnittstelle als Hierarchie von Ressourcen
  - intermediary: agiert sowohl als Client, als auch als Server; leitet Requests und Responses weiter bzw. modifiziert sie; Gateway oder Proxy;
- Leichtes Einführen von Zwischenkomponenten möglich durch selbstbeschreibende Nachrichten, generische Client- und Server-Schnittstellen und zustandslose Kommunikation. Keine einzige Komponente braucht Überblick über ganzes System.
- REST ermöglicht Verbindung zu anderen nicht-REST Systemen, indem diese eine REST-konforme Schnittstelle bereitstellen.
- Abbildung REST connectors and components

3.3 Implementierung von RESTful APIs

- APIs die dem REST Archtekturstil folgen werden RESTful genannt
- 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

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 Schemaß
    - 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 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
  - 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. Der Server kann für jedes Feld Parameter anbieten, welche die Auswertung des Feldwertes verändern. Über solche Feldargumente kann Geschäftslogik, aber auch allgemeine Abfragefunktionen, wie z. B. Sortieren und Filtern zur Verfügung gestellt werden.

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. 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 werden soll, anzugeben. Über sogenannte ‘includes’ lassen sich bei JSON:API verbundene Objekte in die Antwort einbeziehen. 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.

5.2 Versionierung und Weiterentwicklung

- 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

5.3 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
- mehr Daten um Request genauer zu machen, sinnvoll um deutlich weniger Daten als Response zu erhalten

5.4 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
  - ermöglicht HTTP cache proxies
  - je spezieller der query string, desto weniger cache Treffer
- GraphQL
  - POST response wird normalerweise nicht gecacht
  - standardmäßig keine ID für caching vorhanden. Empfehlung: API sollte ID pro Objekt bereitstellen

5.5 Batching/Deduping

5.6 Fehlerbehandlung

5.7 Sicherheit

5.8 Kosten

5.9 Lernkurve, Fehlersuche, Community

5.10 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