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: 4. 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.
- 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
- 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)

- 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