Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

placeholder

1.2 Zielsetzung

placeholder

Kapitel 2

Grundlagen

2.1 Erklärung API

Der Begriff API steht für Application Programming Interface (auf Deutsch Anwendungs-Programmier-Schnittstelle). Die Grundlagen heutiger APIs wurden 1952 von David Wheeler, einem Informatikprofessor an der Universität Cambrigde, in einem Leitfaden verfasst. Dieser Leitfaden beschreibt das Extrahieren einer Sub-Routinen-Bibliothek mit einheitlichem dokumentierten Zugriff. [4] Ira Cotton und Frank Greatorex erwähnten den Begriff erstmalig auf der Fall Joint Computer Con.[2] Dabei erweiterten sie den Leitfaden von David Wheeler um einen wesentlichen Punkt: Der konzeptionellen Trennung der Schnittstelle und Implementierung der Sub-Routinen-Bibliothek. Somit kann die Implementierung auf die die Schnittstelle zugreift ohne Einfluss auf die Benutzer ausgetauscht werden.[3]

> APIs sind wie User Interfaces - nur mit anderen Nutzern im Fokus. David Berlind [1]

APIs sind also wie UIs für die Interaktion mit Benutzern gedacht. Der wesentliche Unterschied zwischen UI-Schnittstellen und einer API liegt aber an der Art der Nutzer die auf das Interface zugreifen. Bei UIs spricht man von einem *human-readable-interface*, das bedeutet das ein menschlicher User mit dem System interagiert. Bei einer API spricht man von einem *machine-readable-interface*, also von einer Schnittstelle die für die Kommunikation zwischen Maschinen gedacht ist.

2.1.1 Abstraktes Beispiel API

Ein abstraktes Beispiel für eine API wäre beispielsweise die Post. Angenommen eine Person will einen Brief an eine andere Person senden, um diese Person zum Essen einzuladen. Was passiert ist dann folgendes:

- Person 1 verfasst eine Nachricht und packt diesen in einen Briefumschlag
- Der Briefumschlag wird mit einer Briefmarke und der Adresse des Empfängers und des Absenders versehen
- Der Brief wird nun an die Post übergeben
- Die Post kümmert sich um die Zustellung des Briefes

- Person 2 erhält den Brief

Damit dieser Zugriff der unterschiedlichen Systeme (hier Menschen) auf die Post funktioniert muss eine genaue Definition des Service vorliegen. Die genaue Spezifikation des Service ist dabei folgende:

- Das zu versendende Objekt (Brief, Paket, etc.) muss an einem Sammelposten der Post abgegeben werden
- Das Objekt ist dabei mit einer Empfängeradresse und einer Absenderadresse zu versehen, zusätzlich muss eine Briefmarke gekauft werden

Das stellt die Art des Services dar und legt somit fest was über die API versendet wird. Zudem wird die Repräsentation der API definiert, also in welcher Form der Service im System des Servicenutzers integiert wird. Der Briefkasten / Sammelposten ist dabei die eigentliche Schnittstelle - also die API. Die Zustellung ist dabei Implementierungsdetail, der Weg vom Sammelposten der Post über die Verteilerzentren und mit dem Briefträger zum Empfänger. Dieses Implementierungsdetail kann beliebig angepasst werden, die Zustellung kann beispielsweise über den Land- oder Luftweg erfolgen. Der Benutzer welcher den Brief versendet hat ist davon nicht betroffen.

2.2 Erklärung REST-API

REST steht für Representational State Transfer [4]. REST ist dabei aber keine konkrete Technologie oder ein Standard. REST beschreibt einen Architekturstil welcher im Jahr 2000 von Roy Fielding konzipiert wurde. Bei REST werden Daten als Ressourcen gesehen und in einem spezifischen Format übertragen. Ursprünglich wurde von Fielding dabei XML verwendet. XML wird aber in den letzten Jahren verstärkt durch JSON abgelöst.

Deswegen ist JSON besser als XML für den Austausch einfacher Daten mittels einer API geeignet:

- JSON wurde speziell für den leichtgewichtigen Datenaustausch konzipiert.
- JSON ist schneller als XML
- JSON hat weniger Overhead als XML da XML deklarativ ist und somit wesentlich mehr Daten als JSON beinhaltet

Zum Vergleich hier ein Buch welches in JSON und XML repräsentiert wird:

```
1 {
2     "isbn": "978-3551555557",
3     "title": "Harry Potter und der Orden des Phönix",
4     "releaseDate": "2003-11-15T00:00:00.000Z"
5 }
```

```
1 <book>
2     <isbn name="isbn" type="string">978-3551555557</isbn>
3     <title name="title" type="string">Harry Potter und der Orden des Phönix</title>
4     <releaseDate name="isbn" type="dateTime">2003-11-15T00:00:00.000Z</releaseDate>
5 </book>
```

Wie in den obigen Code Beispielen ersichtlich produziert JSON (134 Bytes) wesentlich weniger Daten als XML(238 Bytes). Man kann zusammengefasst also sagen, wenn

es um die reine Datenübertragunsrate geht, ist JSON deutlich leichtgewichtiger und damit auch performanter.

2.2.1 REST-Bedingungen

Die Begriffe Web und HTTP müssen von REST unterschieden werden da REST lediglich einen Architekturstil beschreibt. Das Web hingegen besteht aus mehreren Architekturstilen und nutzt als Standargkommunikationsprotokoll HTTP. HTTP ist als REST-konforme Implementierung zu erachten.

1. **Einheitliche Schnittstelle:** Jeder REST-Client der auf den Server zugreift muss auf dieselbe Art und Weise zugreifen. Dabei ist es egal ob der Client ein Browser, eine Desktop-Applikation oder eine mobile Anwendung ist. Dabei kommt hier HTTP mit der Verwendung einer URI - Unique Resource Identifier, welche dann eine URL ist wie zum Beispiel: *https://api.twitter.com* ist zu tragen. Desweiteren müssen Anfragen autark sein - der Client überträgt alle Informationen, die der Server für die Abarbeitung der Anfrage benötigt, mit. Zudem wird die Implementierung von dem Service der sie zur Verfügung stellt entkoppelt, man kann also die Implementierung beliebig weiterentwickeln oder austauschen.
2. **Client-Server Architektur:** Durch das Client-Server Prinzip werden verteilte Systeme beschrieben, die mittels netzwerkbasierter Kommunikation miteinander kommunizieren. In der REST-Architektur sind der Client und der Server klar voneinander abgetrennt. Damit ist es möglich beide Komponenten unabhängig voneinander weiterzuentwickeln, ohne dabei Einfluss auf die jeweils andere Kompnente zu haben
3. **Zustandslosigkeit (Stateless):** Zustandlosigkeit bedeutet, dass der Client alle Informationen die der Server benötigt mitsendet. Das resultiert darin, dass der Server keine Overhad Daten des Clients speichern muss um zukünfige Anfragen abarbeiten zu können. Dadurch ist es auch möglich eine Lastverteilung (also mehrere Serverinstanzen nebeneinander laufen zu lassen) zu realisieren. Dies wiederum erleichtert die Skalierbarkeit des Servers.
4. **Schichtsystem:** Geschichtete Systeme sind Systeme die aus mehreren hierarchischen angeordneten Schichten bestehen. Ein geschichtetes System wäre in .NET beispielsweise so umzusetzen:
   - Controller (Leitet die Anfrage an die Logik weiter)
   - Business-Logik (Setzt die geforderte Aktion um)
   - Datenbankzugriff

   Durch dieses Schichtsystem werden die Abhängikeiten im System reduziert und die Austauschbarkeit der einzelnen Komponenten erleichtert.
5. **Zwischenspeicher (Cache):** Um wiederkehrende Anfragen performanter abwickeln zu können, kann dem Server mithilfe des *Cache-Control-Headers* mitgeteilt werden, ob die angefragten Daten zwischengespeichert werden sollen. Zusätzlich muss definiert werden wielange dieser Cache gültig ist. Nach Ablauf der Gültigkeit werden die Daten erneut vom Server geladen. Durch das Zwischenspeichern von Daten wird die Performanz des Servers gesteigert. Ein Nachteil besteht dahinge-

hend, dass dem Client womöglich nicht immer die aktuellsten Daten zur Verfügung stehen.

6. **Code auf Anfrage:** Hierbei handelt es sich um eine optionale Bedingung, die Code-Snippets für die lokale Ausführung an den Client senden kann. Beispielsweise als JavaScript-Code innerhalb einer HTML-Antwort.

2.2.2 Kommunikation mit dem Server

Die Kommunikation zwischen Server und Client wird bei einer RESTful-API mittels HTTP und dessen Methoden realisert. Um CRUD (Create, Read, Update, Delete) Operationen auf Ressourcen abzusetzen werden folgende Methoden verwendet:

1. **GET:** Wird als lesender Zugriff auf eine Ressource verwendet.
2. **PUT:** Wird verwendet um eine Ressource zu aktualisieren.
3. **POST:** Wird verwendet um eine neue Ressource zu erstellen.
4. **DELETE:** Wird für das Löschen einer Ressource verwendet.

Zusätzlich wird dem Client mittels HTTP-Statuscodes mitgeteilt ob die Anfrage erfolgreich war oder fehlgeschlagen ist. Beispielsweise werden hier Statuscodes angeführt die an den Client zurückgesendet werden:

- **Erfolgreich:** 200
- **Anfrage fehlerhaft:** 400
- **Server interner Fehler:** 500

2.3 GraphQL

2015 veröffentlichte Facebook unter der MIT-Lizenz GraphQL. GraphQL ist die Spezifikation einer plattformunabhängigen Query-Sprache. GraphQL dient als Übersetzer der Kommunikation zwischen Client und Server. Die Kommunikation erfolgt wie bei REST über ein Request-Response-Schema. Der wesentliche Unterschied zu REST-APIs besteht darin, dass GraphQL genau einen Endpunkt, anstatt für jede Ressource einen eigenen Endpunkt, zur Verfügung zu stellt. Desweiteren werden nur POST-Anfragen von GraphQL akzeptiert, diese können lesend (Query) oder schreibend (Mutation) sein.

2.3.1 Entwurfsprinzipien

GraphQL definiert keine Implementierungsdetails sondern diese Entwurfsprinzipien:

1. **Produktzentriert:** Die Anforderungen der Clients (Darstellung der Daten) stehen im Mittelpunkt. GraphQL bietet mit der Abfragesprache dem Client die Möglichkeit genau die Daten abzufragen, die er tatsächlich benötigt. Die Hierarchie der Abfrage wird dabei durch eine Menge ineinander geschachtelter Felder abgebildet.
2. **Hierarchisch:** Jede Anfrage ist wie die Daten die er anfordert geformt. Das bedeutet, dass der Client die Daten genau in dem Format erhält wie in der Anfrage spezifiziert. Das ist ein intuitiver Weg für den Client um seine Datenanforderungen zu definieren.

3. **Strenge Typisierung:** Jeder GraphQL-Service definiert ein für das System spezifisches Typ-System. Anfragen werden im Kontext dieses Typ-Systems ausgeführt. Mit Tools wie zum Beispiel GraphiQL kann man vor Ausführung der Anfrage sicherstellen, dass sie syntaktisch und semantisch korrekt sind.
4. **Benutzerdefinierte Antwort:** Durch das Typ-System veröffentlicht der GraphQLService die Möglichkeiten, die ein Client hat um auf die dahinterliegenden Daten zuzugreifen. Der Client ist dafür verantwortlich genau zu spezifizieren wie er diese Möglichkeiten nutzen will. Bei einer normalen REST-API liefert ein Endpunkt jene Daten einer Ressource zurück welche vom Server definiert wurden. Bei GraphQL werden hingegen genau jene Daten einer Ressource zurückgegeben, die vom Client in seiner Anfrage definiert wurden.
5. **Introspektion:** Das Typ-System eines GraphQL-Services kann direkt mit der GraphQL-Abfragesprache abgefragt werden. Diese Abfragen werden für die Erstellung von Tools für GraphQL benötigt.

2.4 GraphQL vs. REST

2.4.1 Endpunkte

Eine REST-API bietet für jede Ressource verschiedene Endpunkte an um CRUD Operationen für die jeweilige Ressource auszuführen. GraphQL hingegen bietet nur einen Endpunkt. An diesem kann der Client eine Abfrage mit einer Query oder einer Mutation senden um auf die jeweilige Ressource zuzugreifen.

2.4.2 Over-fetching und Under-fetching

Als Over-fetching wird das Laden von zuvielen oder nicht benötigten Daten bezeichnet. Under-fetching bedeutet, dass man mehr Daten benötigt als der Server dem Client zurückgibt. Dieses Problem tritt bei REST-APIs auf die viele verschiedene Clients mit Daten versorgen müssen (beispielsweise eine Desktop-Applikation, eine Mobile-Anwendung und ein Web-Client). Grundsätzlich wollen alle drei Clients diesselbe Ressourcen abfragen, mit dem Unterschied, dass die Mobile Anwendung beispielsweise nicht alle Daten benötigt. Die Desktop-Applikation möchte aber alle Daten einer Ressource bekommen um diese darzustellen. Eine Lösung dafür wäre beispielsweise einen Endpunkt für jeden Client zu definieren um die für ihn benötigen Daten zur Verfügung zu stellen. Dies resultiert aber in mehr Programmieraufwand und damit auch einer höheren Komplexität der Anwendung.

Dieses Problem kann bei GraphQL nicht auftreten da der Client in seiner Anfrage genau die Daten definiert die er braucht.

2.4.3 Was ist für welches Szenario besser geeignet?

Möchte man ein System entwickeln auf welches verschiedene Clients (die verschiedene Anforderungen an die Ressourcen haben) zugreifen, dann ist GraphQL die bessere Wahl. Würde man dieses System mit einer REST-API umsetzen, müsste man entweder

Probleme mit Under-fetching und Over-fetching in Kauf nehmen, oder für jeden Client eine eigene Route definieren. GraphQL ist in diesem Szenario deshalb als besser zu erachten, weil man sich mit der einmaligen Implementierung des Schemas zusätzliche Routendefinitionen erspart. Dadurch hat man automatisch weniger Entwicklungsaufwand und generell weniger Komplexität.

Verfolgt man aber das Ziel ein System zu entwickeln, welches keine komplexen Daten enthält und beispielsweise nur mit einem Web-Client kommuniziert, ist eine REST-API die bessere Wahl.

Kapitel 3

GraphQL

3.1 Interfaces

3.2 Input Typen

placeholder

3.3 Parameter

3.4 Skalare

placeholder

3.5 Querys

3.5.1 Designempfehlungen

placeholder

3.5.2 Verschachtelte Querys

3.5.3 Variablen

placeholder

3.5.4 Fragmentierung

3.5.5 Aliase

placeholder

3.6 Mutationen

3.6.1 Designempfehlungen

placeholder

3.7 Bekannte Probleme

3.7.1 Authentifizierung, Autorisierung und Rollenmanagement

placeholder

3.7.2 1 + n Problem

3.7.3 Subscriptions

placeholder

placeholder

3.7.4 Fehlermanagement

placeholder

3.7.5 Pagination

3.7.6 Caching

placeholder

Kapitel 4

Entwicklung

4.1 Anwendungsszenario

4.2 Architektur

placeholder

placeholder

4.3 Entwurf Schema

placeholder

4.4 Umsetzung GraphQL mit .NET

4.4.1 Verwendete Bibliotheken

placeholder

4.4.2 Entity Framework

placeholder

placeholder

4.4.3 Umsetzung Authentifizierung, Autorisierung, Rollenmanagement

placeholder

4.4.4 Umsetzung Subscriptions

placeholder

4.4.5 Umsetzung 1 + n Problem

placeholder

4.4.6 Umsetzung Pagination

Kapitel 5

Conclusio

5.1 Fazit

5.2 Ausblick

Literatur

[1] David Berlind. *APIs Are Like User Interfaces–Just With Different Users In Mind*. 2017 (siehe S. 5).

[2] Ira W Cotton und Frank S Greatorex Jr. „Data structures and techniques for remote computer graphics“. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. 1968, S. 533–544 (siehe S. 5).

[3] Dominik Kress. *GraphQL: Eine Einführung in APIs mit GraphQL*. dpunkt. verlag, 2020 (siehe S. 5).

[4] David J Wheeler. „The use of sub-routines in programmes“. In: *Proceedings of the 1952 ACM national meeting (Pittsburgh)*. 1952, S. 235–236 (siehe S. 5, 6).