HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

ZEISS

Vergleich von Introspected REST mit alternativen Ansätzen für die Entwicklung von Web-APIs hinsichtlich Performance, Evolvierbarkeit und Komplexität

Bachelorarbeit

zur Erlangung des akademischen Grades Bachelor of Science im Studiengang Informatik der Hochschule für Technik, Wirtschaft und Kultur Leipzig, Fakultät Informatik und Medien

vorgelegt von

Florian Gerlinghoff

Leipzig, 18. September 2020

Erstgutachter: Prof. Dr. Thomas Riechert

Zweitgutachter: Marcus Kohnert, Dipl.-Ing. (BA)

Kurzzusammenfassung

Introspected REST ist ein neuer Ansatz für die Entwicklung von Web-APIs, welcher auf REST aufbaut. In der vorliegenden Arbeit wird dieser neue API-Stil mit REST, GraphQL und gRPC hinsichtlich der Performance, Evolvierbarkeit und Komplexität bzw. Benutzbarkeit verglichen. Die Performance der untersuchten Introspected-REST-API liegt dabei im gleichen Bereich wie die der REST-API, welche ihrerseits von gRPC und GraphQL übertroffen wird. Auch die Evolvierbarkeit ist gleich gut im Vergleich zu REST. Die Verwendung von Introspected REST trägt vor allem zu einer besseren Benutzbarkeit der API bei.

Abstract

Introspected REST is a new approach to the development of web APIs. It builds upon the REST architectural style. In this thesis, Introspected REST is compared to REST, GraphQL, and gRPC in terms of performance, evolvability, and complexity/usability. The results show that the performance of Introspected REST is in the same order of magnitude as the performance of REST. Both are in turn outperformed by gRPC and GraphQL, respectively. The evolvability rates similarly to REST's evolvability, too. Using Introspected REST for an API does most notably improve its usability.

Inhaltsverzeichnis

1. Einleitung

Es ist Bewegung im Web-API-Bereich. Im Jahr 2000 machte sich REST auf, die zu dieser Zeit verbreiteten SOAP-APIs abzulösen. Infolgedessen verstanden und nutzten Entwicklerinnen und Entwickler zwar das HTTP-Protokoll besser, doch vernachlässigten viele dabei einen wichtigen Bestandteil von REST: das Hypermedia-Prinzip. Roy Fielding, welcher REST zuerst beschrieben hat, sah sich deshalb dazu veranlasst, in einem Blogpost[1] darauf hinzuweisen, dass der Einsatz von Hypermedia für REST-APIs obligatorisch ist. Doch auch danach sah man nur wenige sogenannte *Level-3*-REST-APIs [1] oder aber Clients wurden nicht unter Beachtung des Hypermedia-Prinzips entwickelt. Als Facebook im Jahr 2015 GraphQL der Öffentlichkeit präsentierte, sprangen viele Entwicklerinnen und Entwickler schnell auf diesen neuen API-Stil auf und sagten sich von REST los, was in Überschriften wie *GraphQL is the better REST*[2] und *REST is the new SOAP*[3] mündete. Sie schienen enttäuscht zu sein von dem (semantisch diffusen [2]) Begriff „REST“ und eifrig, eine neue Technologie auf den *Gipfel der überzogenen Erwartungen* [3] zu heben. Im gleichen Jahr wurde mit gRPC ein weiter API-Stil vorgestellt und die Auswahl für API-Anbieter noch vergrößert. Die Frage, wie Web-APIs am besten umgesetzt werden sollen, wird also immer noch diskutiert und das Ende der Geschichte ist noch längst nicht erreicht.

2017 ging die Geschichte weiter, als Filippos Vasilakis Introspected REST vorstellte. Er präsentierte diesen neuen Ansatz für Web-APIs als Alternative sowohl zu REST als auch zu GraphQL. Doch stellt Introspected REST wirklich eine praktikable Alternative zu den heutzutage verbreiteten API-Stilen dar? Welche Vorteile bietet Introspected REST und was kann man daraus lernen? Und sollte ich, als Anbieter einer API, noch heute auf Introspected REST umsteigen? Diese Fragen wurden bisher nirgendwo untersucht, geschweige denn beantwortet. In der vorliegenden Arbeit wird deshalb ein Vergleich von Introspected REST und den oft verwendeten API-Stilen REST, GraphQL und gRPC angestellt. Das Ziel der Arbeit ist, die aufgeworfenen Fragen in einem begrenzten Rahmen zu beantworten.

Der Hauptteil dieser Arbeit beginnt mit einer Einführung in das Gebiet der Web-APIs. Dazu werden die drei API-Stile vorgestellt, welche neben Introspected REST als Vergleichsobjekte herhalten: REST, GraphQL und gRPC. Die Erläuterungen zu REST bilden gleichzeitig eine wichtige Grundlage für das Verständnis von Introspected REST. Dieser neue API-Stil, welcher auf REST aufbaut, wird in Kapitel 3 beschrieben.

[1] R. T. Fielding. (20. Okt. 2008). „REST APIs must be hypertext-driven,“ URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 09.09.2020).

[2] T. Suchanek. „GraphQL is the better REST,“ URL: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/ (besucht am 09.09.2020).

[3] P. de Bonchamp. (15. Dez. 2017). „REST is the new SOAP,“ URL: https://www.freecodecamp.org/news/rest-is-the-new-soap-97ff6c09896d/ (besucht am 09.09.2020).

Der Fokus liegt dabei auf dessen beiden Grundpfeilern: Dem Introspection-Prinzip und Microtypes. Das Introspection-Prinzip beschränkt die Art und Weise, wie Hypermedia zum Client gelangen darf. Es besagt, dass Nutzdaten und Metadaten getrennt werden sollen. Microtypes sind kleine, abgeschlossene Einheiten, welche eine Funktionalität der API beschreiben. Zusammen übernehmen mehrere Microtypes die Aufgabe von Mediatypes in REST; Mediatypes werden in Introspected REST also in Microtypes zerlegt. Im Rahmen des Vergleichs wurde auch ein Prototyp einer Introspected-REST-API entworfen. Details dazu werden ebenfalls in diesem Kapitel ausgeführt.

Vasilakis hat einige Probleme des REST-Stils identifiziert, auf welche in Abschnitt 3.1.2 genauer eingegangen wird. Die genannten Probleme beeinflussen die Faktoren Performance, Evolvierbarkeit und Komplexität. Deshalb werden die API-Stile nach ebendiesen drei Kriterien verglichen. In Kapitel 4 werden die drei Kriterien definiert und Methoden für die Bewertung der API-Stile betrachtet. Für den Vergleich der Performance wird ein Experiment durchgeführt, in welchem API-Clients Aufgaben eines Szenarios bearbeiten müssen. Währenddessen wird die Zeit, welche für die Bearbeitung der Aufgabe benötigt wird, gemessen. Die Evolvierbarkeit wird mithilfe einer ähnlichen Methode bewertet. Es werden Änderungen an einer bestehende API beschrieben. Für jeden API-Stil wird untersucht, welche Anpassungen am Client eine solche Änderung nach sich zieht. Der Vergleich der Komplexität erfolgt durch eine Heuristische Evaluation. Dafür werden Heuristiken für die Benutzbarkeit von APIs sowie allgemeine Usability-Prinzipien aufgegriffen und auf die API-Stile angewendet.

Die Ergebnisse der Vergleiche werden in Kapitel 5 dargelegt. Im Anschluss werden in Kapitel 6 die Ergebnisse ausgewertet und begründet. Vor allem interessiert hier, wie sich das Introspection-Prinzip und Microtypes auf Performance, Evolvierbarkeit und Komplexität auswirken. Kapitel 7 bildet den Abschluss der Arbeit. Hier werden die eingangs gestellten Fragen wieder aufgegriffen und anhand der Ergebnisse der Vergleiche teilweise beantwortet. Zum Schluss werden Fragen aufgeworfen, welche durch diese Arbeit noch offengeblieben oder neu hinzugekommen sind.

Messdaten, Quellcode und sonstige Dateien sind auf dem beiliegendem Datenträger sowie im Repository zur Arbeit[4] zu finden.

[4]BachelorThesis (Repository): https://github.com/Flogex/BachelorThesis

2. Grundlagen von Web-APIs

APIs (Application Programming Interfaces, deutsch: *Programmierschnittstellen*) sind ein wesentlicher Bestandteil der modernen Softwareentwicklung. Sie ermöglichen es, Services für andere Entwicklerinnen und Entwickler zur Verfügung zu stellen, welche wie Lego-Bausteine zusammengesetzt werden können. Somit tragen APIs zu einer effizienteren Softwareentwicklung bei. In diesem Kapitel werden die Grundlagen vorgestellt, die zum Verständnis der darauffolgenden Kapitel relevant sind. Zuerst erfolgt eine allgemeine Einführung von Web-APIs. Im Anschluss daran werden die drei heutzutage hauptsächlich verwendeten API-Stile vorgestellt: REST, GraphQL und gRPC [4, S. 300].

2.1. Definition und Einteilung

Eine **API** ist eine Schnittstelle eines Softwaremoduls oder -systems, welche es anderer Software ermöglicht, mit diesem Modul oder System zu interagieren. APIs abstrahieren die Interna der bereitstellenden Software [5, S. 1]. Sie ermöglichen die Wiederverwendung von Funktionalität.

Web-APIs sind APIs, welche von Webservern bereitgestellt werden [6]. Sie folgen dem Client-Server-Prinzip, verbinden also zwei eigenständige Softwaresysteme lose miteinander. Eine Web-API ermöglicht es dem Client, mit dem Server zu kommunizieren und so die vom Server bereitgestellte Funktionalität zu nutzen. Die API legt dabei fest, *wie* beide Parteien miteinander kommunizieren, bildet also einen Vertrag zwischen beiden [7, S. 4–5]. Meist erfolgt die Kommunikation zwischen Client und Server via HTTP. Wenn nicht anders erwähnt, bezieht sich der Begriff „API“ im Folgenden immer auf Web-APIs. Im weiteren Sinne umfasst der Begriff manchmal auch das gesamte Backend.

Web-APIs lassen sich nach dem verwendeten Interaktionsmuster in Request-Response-APIs und Event-Driven-APIs einteilen. Bei **Request-Response-APIs** definiert der Server Endpunkte, welche durch eine URL identifiziert werden. Ein Client sendet eine Anfrage an einen dieser Endpunkte und der Server gibt eine Antwort zurück. Derartige APIs lassen sich mit REST, RPC und GraphQL umsetzen [5, S. 9]. **Event-Driven-APIs** informieren einen Client, sobald ein Ereignis aufgetreten ist, z.B. eine Änderung an einem Datensatz. Der Server sendet potentiell unendlich viele Antworten an den Client [5, S. 19]. GraphQL [8, S. 31] und gRPC [9, S. 2] bieten von Haus aus die Möglichkeit, Event-Driven-APIs zu erstellen. In dieser Arbeit werden nur Request-Response-APIs betrachtet.

Eine weitere mögliche Unterteilung ist nach der Zielgruppe in Public-APIs und Private-APIs. **Public-APIs** sind für fast jeden zugänglich und nutzbar und es gibt

meist keine vertraglichen Regelungen zwischen dem Anbieter und dem Nutzer bis auf die Nutzungsbedingungen. **Private-APIs** hingegen sind nur für interne Angestellte und ggf. Partner des Anbieters zugänglich [7, S. 7]. Laut Mitra ist das Ziel für Anbieter von Public-APIs eine weite Verbreitung und häufige Nutzung der API. Private-APIs sollen vor allem zu Kosteneinsparungen führen, z.B. bei der Entwicklung neuer Endanwendungen [10, 06:47–07:03]. Nach Jacobson et al. können durch Private-APIs vor allem höhere Effizienz bei der Entwicklung von Anwendungen und eine bessere Nutzung des vorhandenen immateriellen Vermögens, z.B. des Datenbestands, erreicht werden [7, S. 27].

2.2. Der Architekturstil Representational State Transfer (REST)

Representational State Transfer (REST) ist ein Architekturstil für verteilte Hypermedia-Systeme, welcher im Rahmen der Spezifikation des HTTP-Protokolls erarbeitet wurde und als Richtlinie für die Weiterentwicklung des World Wide Webs diente [11]. Später wurde REST auf Web-APIs übertragen und ist heute der am weitesten verbreitete API-Stil [5, S. 10].

In diesem Abschnitt werden die grundlegenden Bausteine des Webs – und damit auch von REST [12, S. 12] und Introspected REST – erläutert. Danach werden die REST-Prinzipien eingeführt, welche obligatorisch für eine REST-Architektur sind. Zum Schluss wird die Interaktion zwischen Client und Server beschrieben.

2.2.1. Grundlagen des Webs

Das World Wide Web (kurz: Web) ist ein verteiltes Hypertext-System nach dem Client-Server-Prinzip [13]. Durch Hypermedia verbindet es Ressourcen, welche durch eine URI identifiziert werden, untereinander.

Ressourcen und Repräsentationen

Das Web ermöglicht den Zugriff auf Ressourcen eines Servers über standardisierte Protokolle. Berners-Lee et al. definieren **Ressourcen** als „items of interest“ [14, Kap. 1], d.h. beliebige konzeptuelle oder physische Dinge, auf welche verwiesen oder zugegriffen werden soll. Beispiele für Ressourcen sind eine Webseite, ein Gebäude, die Farbe Schwarz und die Beziehung zwischen Mutter und Kind. Da Ressourcen von zentraler Bedeutung für das Web sind, wird dieses auch als *ressourcenorientiert* bezeichnet. Selbiges gilt für REST-Architekturen [12, S. 4].

Eine Ressource wird identifiziert durch einen **Uniform Resource Identifier (URI)**. Eine URI ist eine Zeichenfolge, welche der Grammatik in RFC 3986 entspricht [15]. URIs ermöglichen das Teilen von Informationen mit Dritten [14, Kap. 2] sowie den

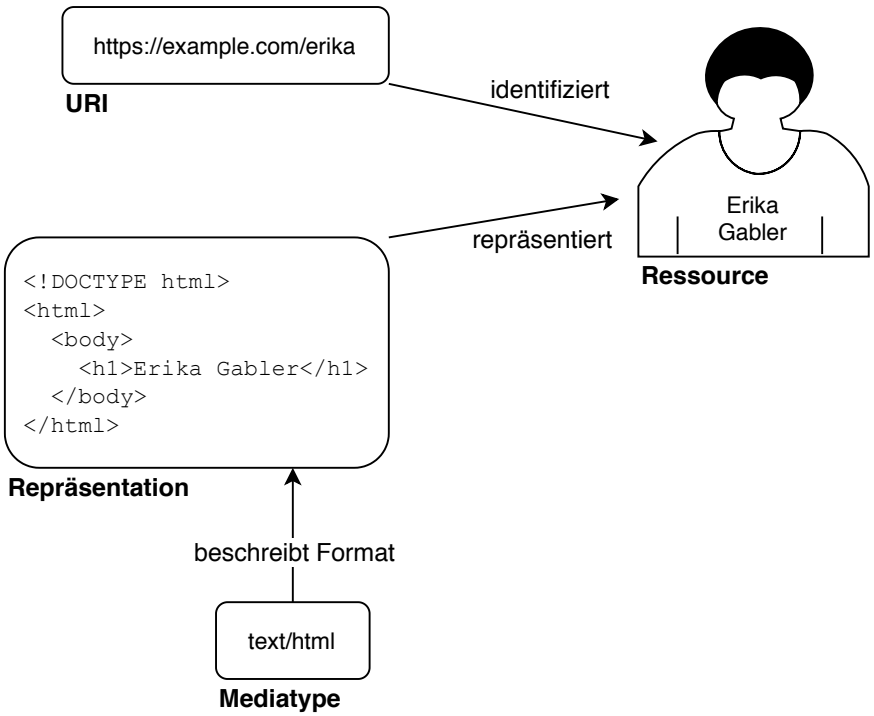

Abbildung 2.1.: Beziehung zwischen Ressource, URI, Repräsentation und Mediatype (nach [14, Kap. 1])

Verweis auf andere Ressourcen, eingebettet in eine Repräsentation, durch Hyperlinks [14, Abs. 4.4].

Wenn ein Client einer URI folgt, erhält er eine **Repräsentation**. Eine Repräsentation enthält Informationen über die durch die URI identifizierte Ressource [14, Abs. 3.2] oder andere Informationen wie etwa eine Fehlermeldung [16, S. 91]. Da eine Ressource mehrere Repräsentationen haben kann, können verschiedene Clients durch Content-Negotiation die für sie geeignetste Repräsentation auswählen [17, Abs. 3.4].

Das Datenformat einer Repräsentation wird in einem **Mediatype** spezifiziert [18]. Um eine Repräsentation verarbeiten zu können, muss ein Client den zugrundeliegenden Mediatype kennen [19, S. 358]. HTML bzw. *text/html* ist ein Mediatype für strukturierte Dokumente [20]. JSON bzw. *application/json* ist ein Mediatype für den Datenaustausch [21] und findet vor allem bei Web-APIs Verwendung. Abbildung 2.1 verdeutlicht das Zusammenspiel von Ressourcen, URIs, Repräsentationen und Mediatypes.

Hypermedia

Hypermedia bezeichnet Metadaten, welche Beziehungen zwischen Ressourcen beschreiben und dem Client dadurch Informationen über die Interaktionsmöglichkeiten

mit dem Server geben [19, S. 45]. Hypermedia wird ausgedrückt durch Hypermedia-Elemente, welche im verwendeten Mediatype spezifiziert sind. Wird ein Hypermedia-Element ausgelöst, findet eine Interaktion mit dem Server statt. Die Semantik der Interaktion wird durch das Hypermedia-Element definiert. Beispielsweise kann dieses implizite oder explizite Möglichkeiten bieten, um das Link-Modell von RFC 8288 ([22, Abs. 2]) abzubilden. So impliziert ein `<a>`-Tag in HTML einen HTTP-GET-Request, wobei *Link-Relation-Type* und *Link-Target* durch die HTML-Attribute `rel` und `href` explizit festgelegt werden.

Nach Richardson und Amundsen erfüllen Hypermedia-Elemente drei Aufgaben:

- Sie geben einem Client Hinweise zum Senden eines Requests. Beispielsweise legt der `<a>`-Tag in HTML die HTTP-Methode und die URL des Requests fest.
- Sie geben einem Client Hinweise zur zu erwartenden Antwort. Beispielsweise kann der `<a>`-Tag den Mediatype der Antwort spezifizieren.
- Sie teilen einem Client mit, wie er die Antwort in seinen Arbeitsablauf integrieren kann. Das bedeutet, dass der Server dem Client durch Hypermedia-Elemente Hinweise gibt, welche Requests er als nächstes Stellen könnte [19, S. 52].

Der letzte Punkt beschreibt Hypermedia-Elemente als *Affordances* (deutsch: *Angebotscharakter*), durch welche sich ein Client für die nächste Aktion entscheiden kann [23]. Gibson beschreibt Affordances wie folgt, ohne dabei das Web im Sinn zu haben:

> The *affordances* of the environment are what it *offers* […], what it *provides* or *furnishes*, either for good or ill. [24, S. 127, Hervorheb. im Original]

2.2.2. REST-Prinzipien

REST definiert sechs Prinzipien (engl.: *Constraints*) für die Architektur verteilter Hypermedia-Anwendungen. Diese wurden von Roy T. Fielding formuliert, um durch die Softwarearchitektur Eigenschaften zu realisieren, welche für ein System ähnlich des Webs wünschenswert sind [19, S. 341]. Diese erwünschten Eigenschaften sind:

- eine niedrige Einstiegshürde, um eine ausreichende Anzahl an Nutzern zu erreichen
- Erweiterbarkeit, um auf sich verändernde Anforderungen reagieren zu können
- Nutzung von Hypermedia, damit nicht nur Daten, sondern auch Steuerinformationen, welche Zustandsübergänge und Ressourcenmanipulationen beschreiben, vom Server verwaltet werden können
- Skalierbarkeit, um Informationen über Organisationsgrenzen hinweg austauschen zu können [16, S. 66–71; 19, S. 342–344]

Die REST-Prinzipien wurden später auf Web-APIs angewendet. Während sich aber die erwünschten Eigenschaften im Web gegenseitig ergänzen, stehen sie bei Web-APIs im Gegensatz zueinander, denn das Web ist für Maschine-zu-Mensch-Kommunikation

gedacht, Web-APIs aber für die Machine-zu-Maschine-Kommunikation. Für Maschinen ist es schwieriger, die Bedeutung von Hypermedia-Elementen zu verstehen, und ein Mensch wird benötigt, um zu entscheiden, welcher Zustandsübergang gewählt werden soll. Die Nutzung von Hypermedia erhöht folglich die Einstiegshürde. Der Verzicht von Hypermedia geht indes auf Kosten der Erweiterbarkeit und Skalierbarkeit [19, S. 344f.].

Entwicklerinnen und Entwickler von öffentlichen APIs mit vielen nicht unter derer Kontrolle stehenden Clients müssen sich entscheiden zwischen Einfachheit und Erweiterbarkeit. Einfachheit erreichen sie, indem Steuerinformationen durch einen Menschen anhand einer Dokumentation im Client kodiert werden. Erweiterbarkeit erreichen sie durch die Nutzung von Hypermedia. Stehen alle Clients unter ihrer Kontrolle, ist die API nicht skalierbar, dafür kann aber Einfachheit durch Verzicht auf Hypermedia erreicht werden, während Erweiterbarkeit immer noch gewährleistet ist, indem alle Clients auf einmal angepasst werden [19, S. 345].

Im Folgenden werden die REST-Prinzipien beschrieben, welche zur Realisierung der genannten Eigenschaften beitragen.

Client-Server

Eine Client-Server-Architektur besteht aus einem Server, welcher einen oder mehrere Dienste bereitstellt, und einem Client, welcher diese Dienste nutzen kann. Client und Server kommunizieren durch den Austausch von Nachrichten [25, Abs. 1.3.], wobei der Client eine Anfrage (engl.: *Request*) an den Server sendet und dieser eine Antwort (engl.: *Response*) zurückschickt [16, S. 45f., 78].

Zustandslose Kommunikation

Das Prinzip der zustandslosen Kommunikation besagt, dass der Client nicht davon ausgehen darf, dass der Server clientspezifische Informationen über die aktuelle Sitzung verfügt. Stattdessen muss die Anfrage eines Clients selbst alle Informationen beinhalten, welche der Server benötigt, um die Anfrage zu bearbeiten [16, S. 47, 78f.]. Der Zustand der Sitzung wird auf dem Client gespeichert [26, Abs. 2.2]. Jede Anfrage ist unabhängig von allen anderen [19, S. 349].

Caching

Durch das Caching-Prinzip kann ein Cache als zusätzliche Komponente zwischen Client und Server eingefügt werden und die Antworten auf Anfragen eines oder mehrerer Clients speichern kann. Anfragen, für welche der Cache die Antworten gespeichert hat, können durch den Cache beantworten werden. Nur Anfragen, deren Antwort noch nicht gecacht wurde, werden an den Server weitergeleitet [16, S. 44, 48, 79f.]. Dieses Prinzip wird ermöglicht durch zustandslose Kommunikation und selbstbeschreibende

Nachrichten, da die Anfrage des Clients unabhängig von vorherigen Anfragen ist und die gecachte Antwort des Servers alle notwendigen Informationen enthält.

Einheitliche Schnittstelle

Das Prinzip der einheitlichen Schnittstelle besagt, dass Client und Server anhand eines standardisierten Vertrags kommunizieren und sich bspw. an die Semantik des verwendeten Protokolls (etwa HTTP) und des Mediatypes halten. Eine einheitliche Schnittstelle ist nicht ressourcenspezifisch, sondern generisch [23]. Dieses Prinzip impliziert selbst vier weitere Prinzipien:

- ADRESSIERBARKEIT VON RESSOURCEN: Jede Ressource wird durch eine URI identifiziert. Die URI bleibt gleich, auch wenn sich der Zustand der Ressource ändert.
- MANIPULATION VON RESSOURCEN DURCH REPRÄSENTATIONEN: Möchte ein Client eine Ressource auf dem Server verändern, schickt er eine Repräsentation zum Server, welche den gewünschten Zustand der Ressource beschreibt.
- SELBSTBESCHREIBENDE NACHRICHTEN: Eine Nachricht enthält alle Informationen, die notwendig sind, damit der Empfänger die Nachricht verarbeiten kann. Sind Informationen aus anderen Dokumenten zum Verständnis notwendig, enthält die Nachricht einen Link auf diese Dokumente.
- HYPERMEDIA AS THE ENGINE OF APPLICATION STATE: Ein Client bedarf keines vorherigen Wissens über eine API bis auf die Einstiegs-URL und das Verständnis der von der API verwendeten Mediatypes. Alle Informationen, die der Client zur Kommunikation mit dem Server benötigt, werden durch Hypermedia bereitgestellt. Mögliche Zustandsübergänge werden durch den Server angezeigt (Affordances), der Client wählt einen dieser Übergänge [23; 19, S. 345ff.].

Das Prinzip der einheitlichen Schnittstelle ist ausschlaggebend für die Abgrenzung von REST von anderen Architekturstilen [16, Abb. 5-9].

Mehrschichtiges System

Das Prinzip der mehrschichtigen Gesamtsysteme ermöglicht es, dass ein System nicht nur aus einem Server und einem oder mehreren Clients besteht. Stattdessen können sich zwischen diesen beiden Komponenten Vermittler (Proxys und Gateways) befinden, welche sowohl als Client als auch als Server fungieren. Dabei kann eine Komponente einer Schicht nur mit den Komponenten direkt angrenzender Schichten kommunizieren und hat kein Wissen über dahinterliegende Schichten. Die Vermittler können die Nachricht umformen, da diese selbstbeschreibend ist [16, S. 46f., 82ff.].

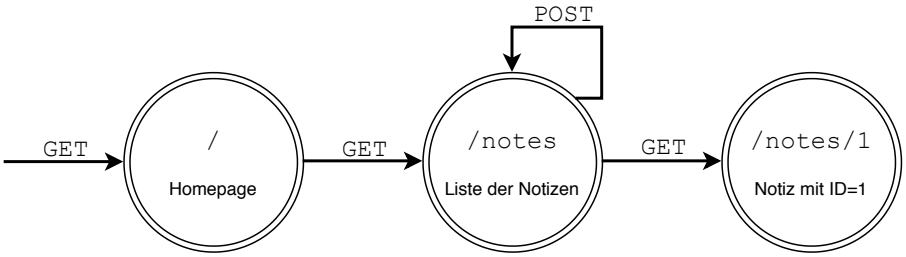

Abbildung 2.2.: Zustandsautomat aus Clientperspektive (nach [19, Abb. 1-7])

Code auf Anforderung

Nach dem Prinzip *Code auf Anforderung* kann ein Server einem Client mobilen Code schicken, welcher die Verarbeitung einer Ressource beschreibt, falls jener weiß, wie die Ressource zu verarbeiten ist. Der Client führt den Code lokal aus. Dieses Prinzip ist für ein REST-System optional [16, S. 53, 84].

2.2.3. Ablauf der Interaktionen zwischen Client und Server

Client und Server in einer REST-API bilden zusammen einen verteilten Zustandsautomaten. Auf dem Client wird der aktuelle Zustand des Automaten gehalten. Der Initialzustand ist die „Homepage“ der API. Ein Client betritt den Automaten durch eine Anfrage an den Homepage-Endpunkt. Die Einstiegs-URL muss dem Client vor der ersten Interaktion mit der API bekannt sein [23]. In jeder Antwort sendet der Server die erlaubten Zustandsübergänge in Form von Hypermedia. Der Client führt einen Zustandsübergang aus, indem er eine Anfrage an einen Endpunkt des Servers sendet. Die Antwort des Servers repräsentiert den neuen Anwendungszustand [19, S. 2–11]. In HTTP enthält die Anfrage neben der URI der angefragten Ressource auch die HTTP-Methode, welche die Semantik der Interaktion beschreibt [12, S. 11].

In Abbildung 2.2 ist die Veränderung des Clientzustands dargestellt. Durch Anfragen an den Server wird ein Zustandsübergang ausgeführt. Der Initialzustand wird durch einen GET-Request an den Homepage-Endpunkt hergestellt.

Um den Zustand einer Ressource zu ändern, sendet der Client eine Repräsentation des gewünschten Zustands (HTTP POST/PUT) oder der Zustandsänderungen (HTTP PATCH) an den Server. Der Client kann den Zustand einer Ressource nicht direkt manipulieren, sondern nur den Server anfragen, den Zustand zu ändern [19, S. 12f.].

Laut Fielding eignet sich REST vor allem für eine grobkörnige Unterteilung der Domäne in Ressourcen [16, S. 101]. Dadurch müssen Clients weniger Requests senden, um die benötigten Daten zu erhalten. Allerdings kann der Server zu viele Daten zurückliefern (Overfetching). Nottingham argumentiert, dass das Multiplexing in

HTTP/2 es ermöglicht, eine effiziente feingliedrige Unterteilung vorzunehmen, d.h. viele kleine, verlinkte Ressourcen anstatt weniger großen zu erstellen [27].

2.3. Die Abfragesprache GraphQL

GraphQL bezeichnet eine Sprache für Web-APIs zur Abfrage und Manipulation von Daten sowie eine Ausführungsumgebung für ebendiese Operationen [28]. GraphQL wurde 2015 von Facebook veröffentlicht. Die Entwicklung startete, um eine Alternative zu REST-APIs und SQL-ähnlichen Abfragesprachen zu schaffen, welche besser auf die Anforderungen der internen mobilen Anwendungen von Facebook zugeschnitten ist [29].

> We were frustrated with the differences between the data we wanted to use in our apps and the server queries they required. We don't think of data in terms of resource URLs, secondary keys, or join tables; we think about it in terms of a graph of objects and the models we ultimately use in our apps like NSObjects or JSON. [29]

GraphQL bietet drei verschiedene Operationstypen. Durch **Querys** können Daten abgefragt und durch **Mutations** geändert werden. Weiterhin kann sich ein Client durch **Subscriptions** über Änderungen an den Daten benachrichtigen lassen. Eine Interaktion zwischen Client und Server besteht für Querys nur aus einem einzigen Request und einem einzigen Response, da diese ausreichen, um alle benötigten Daten zu erhalten. Auch mehrere Mutationen können meist mit einem einzigen Request ausgeführt werden. Für Subscriptions sendet der Client einen Request, bekommt aber, zeitlich versetzt, mehrere Responses zurück.

GraphQL-Operationen sind stark typisiert. Im Folgenden wird das Typsystem von GraphQL kurz vorgestellt. Weitere Details zum Typsystem und GraphQL allgemein können in der GraphQL-Spezifikation ([28]) nachgelesen werden.

Das Typsystem dient sowohl dazu, eine Operation zu validieren, als auch, einen Client über die möglichen Operationen, d.h. über die Fähigkeiten des GraphQL-Servers, zu informieren [28, Abs. 3]. Es existieren skalare Datentypen, Objekte und abstrakte Datentypen. Zu den Skalaren zählen die primitiven Datentypen, benutzerdefinierte Skalare sowie Enumerationen. Objekte beschreiben eine Liste von Feldern. Felder bestehen aus Bezeichner, Typ und optionalen Argumenten. Felder sind Funktionen, welche Daten zurückgeben und bei Mutations zusätzlich Seiteneffekte haben[1]. Argumente können den zurückgegebenen Wert oder den Seiteneffekt beeinflussen.

Alle Typen des GraphQL-Services bilden zusammen das **Schema**. In der Schemadefinition werden die *Root-Operation-Typen* für Querys, Mutations und Subscriptions festgelegt. Ein Root-Operation-Typ selbst ist ein Objekt und verfügt über Felder,

[1] Seiteneffekte sind zwar auch bei Querys möglich, allerdings entspricht dies nicht der Konvention.

welche durch einen Client abgefragt bzw. ausgeführt werden können. Root-Operation-Typen bilden den Einstiegspunkt in die jeweilige Operation.

In einem Query werden, beginnend von dem im Schema festgelegten Root-Query-Typen, die Felder ausgewählt, welche der Client benötigt. Diese Felder bilden das **Selection-Set**. Wird der Query ausgeführt, wird für jedes Feld der entsprechende Wert bestimmt. Ist ein ausgewähltes Feld ein Objekt, muss für dieses selbst ein Selection-Set angegeben werden. Somit ist das Selection-Set ein Baum mit dem Root-Query-Typen als Wurzel. Skalare bilden die Blätter des Baums, während Objekte die inneren Knoten darstellen [28]. Durch die exakte Bestimmung der Daten, welche in Client benötigt, werden Over- und Underfetching vermieden [8, S. 7ff.].

Introspection ermöglicht es, das Schema des GraphQL-Servers, d.h. alle verfügbaren Typen und deren Eigenschaften und Felder, zur Laufzeit abzufragen. Wenn die Felder des Schemas eine Beschreibung erhalten, kann durch Introspection eine Dokumentation generiert werden [28, Abs. 4.2]. Weiterhin ermöglicht Introspection z.B. die Codegenerierung für Clients oder die Unterstützung von GraphQL-Benutzerinnen und -Benutzern in Entwicklungsumgebungen [30, S. 26].

2.4. Das gRPC-Protokoll

gRPC ist ein von Google entwickeltes RPC-Protokoll und gleichzeitig plattformunabhängiges Framework. Es dient dazu, Services innerhalb von und zwischen Datenzentren zu verbinden, kann aber auch für den Zugriff von Endgeräten auf einen Backend-Service eingesetzt werden [31].

Remote Procedure Call (RPC) ist ein Modell, um Prozeduren über ein Kommunikationsnetzwerk aufzurufen. Es wurde als erstes in RFC 707 als *Procedure Call Model* beschrieben [32]. In RPC ruft ein Client eine Prozedur eines Servers auf, indem er eine Nachricht mit dem Namen und den Argumenten der Prozedur sendet. Die Ausführung der aufrufenden Prozedur auf dem Client wird blockiert. Der Server führt daraufhin die aufgerufene Prozedur aus und sendet das Resultat zurück an den Client, welcher die Ausführung fortsetzen kann [33].

Nach Birrel und Nelson bietet RPC eine einfache Semantik, hohe Effizienz und Generalität [34]. Remote-Procedure-Calls sollen wie lokale Prozeduraufrufe aussehen [32]. Idealerweise unterscheidet sich ein Prozeduraufruf über das Netzwerk für Entwicklerinnen und Entwickler nicht von einem lokalen Aufruf. Dadurch verbirgt RPC das Netzwerk und suggeriert Lokalität. Nach Waldo et al. erfordert das Programmieren verteilter Systeme aber, die Latenz des Netzwerks und Probleme wie einen Teilausfall von Netzwerkknoten zu berücksichtigen. Weiterhin muss eine andere Art des Speicherzugriffs gewählt werden. Deshalb gibt es „fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects.“ [35]

gRPC ist ein Framework für RPC. Es wurde 2015 von Google veröffentlicht und ist der Nachfolger von *Stubby*. Stubby ist ein RPC-Framework, welches in Googles Rechenzentren eingesetzt wurde, um eine große Anzahl an Microservices zu verbinden, und Effizienz, Sicherheit und Zuverlässigkeit gewährleisten sollte [36]. gRPC definiert ein Protokoll für die Interprozesskommunikation und übernimmt daneben Authentifizierung, Zugriffskontrolle und Überwachung der Kommunikation [9, S. 3]. gRPC ermöglicht einen bidirektionalen Austausch von Nachrichten, entweder nach dem Request-Response-Schema oder als Stream. Die Kommunikation kann sowohl synchron als auch asynchron stattfinden [9, S. 2]. Als Transportprotokoll wird HTTP/2 verwendet.

Jeder gRPC-Service veröffentlicht die Definition seiner Schnittstelle, welche die vorhandenen Prozeduren mit Argumenten und Rückgabetypen beschreibt. Als Sprache für diese Servicedefinition werden **Protocol Buffers (Protobuf)** verwendet. Diese dienen gleichzeitig als binäres Serialisierungsformat, wobei die Verwendung anderer Formate möglich ist [9, S. 4]. Eine vollständige Beschreibung von Protobuf 3 findet sich im Language Guide[2].

Aus der Servicedefinition kann ein **Client-Stub** generiert werden. Der Client-Stub besitzt die gleiche Schnittstelle wie der gRPC-Service, übertragen in die jeweilige Programmiersprache. Er übernimmt die Serialisierung und Deserialisierung der Nachrichten sowie die Netzwerkkommunikation. Dadurch muss sich eine Entwicklerin oder ein Entwickler nicht mehr selbst um die Details des Nachrichtenaustauschs kümmern und der Aufruf der Prozedur auf dem Server erscheint so einfach wie ein lokaler Prozeduraufruf. Weiterhin kann ein **Server-Skeleton** aus der Servicedefinition generiert werden, welches die Details der Kommunikation auf Serverseite abstrahiert [9, S. 2f.]. Neben dem Stub bzw. Skeleton selbst werden auch die Klassen oder Interfaces für die Nachrichtentypen erzeugt [9, S. 25]. Die Codegenerierung erfolgt durch den Protocol-Buffer-Compiler (protoc) und ist von der gewählten Programmiersprache abhängig. Zum Zeitpunkt des Schreibens werden elf Programmiersprachen unterstützt [37].

Die Interaktionen zwischen Client und Server verhalten sich ähnlich wie die Interaktionen bei der Verwendung einer lokalen Programmbibliothek. Je nach Anwendungsfall reicht ein einziger Request aus, z.B. um eine einzelne Aktion anzustoßen, oder es ist ein komplexeres Hin und Her zwischen beiden Parteien notwendig.

2.5. Zusammenfassung

In diesem Kapitel wurden Web-APIs als Vertrag zwischen Client und Server definiert und nach Interaktionsmustern in Request-Response- und Event-Driven-APIs sowie

[2]Language Guide (proto3): https://developers.google.com/protocol-buffers/docs/proto3 (besucht am 29.08.2020)

nach Zielgruppe in Public- und Private-APIs unterteilt.

REST ist ein Architekturstil für Web-APIs, bei dem Client und Server miteinander kommunizieren, indem der Client eine Anfrage an einen Endpunkt des Servers mit zugehöriger URI stellt. Der Server antwortet mit einer Repräsentation der durch diese URI identifizierten Ressource oder einer Repräsentation anderer Informationen. Die Antwort enthält Hypermedia-Elemente, welche dem Client die nächsten möglichen Anfragen anzeigen, die er stellen kann. Clients können den Zustand von Ressourcen durch Repräsentationen verändern. REST-APIs erfüllen die sechs REST-Prinzipien, wobei das Prinzip der einheitlichen Schnittstelle eine essentielle Rolle spielt.

GraphQL ist eine Sprache für die Abfrage und Manipulation von Daten. Alle Operationen sind stark typisiert und werden durch das Schema beschrieben. Introspection ermöglicht es Clients, das Schema abzufragen und so die API zu erkunden und Requests vorab zu validieren.

gRPC ist ein Protokoll und Framework für Remote-Procedure-Calls und wird vorrangig für Interprozesskommunikation eingesetzt, kann aber auch für Public-APIs verwendet werden. gRPC-Services verfügen über eine Servicedefinition, aus welcher Code für Client und Server generiert werden kann. Der generierte Code verbirgt die Details des Nachrichtenaustauschs. Als Sprache für die Servicedefinition sowie als Datenübertragungsformat werden Protocol Buffers verwendet.

Nachdem nun drei weit verbreitete API-Stile erläutert und vor allem die Grundlagen von REST eingeführt wurden, wird im nächsten Kapitel Introspected REST vorgestellt, ein neuer API-Stil, welcher sich als Alternative zu REST und GraphQL positioniert.

3. Introspected REST

In diesem Kapitel wird Introspected REST vorgestellt, ein neuer Architekturstil, welcher auf REST aufbaut und als Alternative zu REST und GraphQL positioniert wird [38]. Zu Beginn werden die Motivation für Introspected REST dargestellt und die zwei Grundpfeiler, Introspection und Microtypes, erläutert. Im Anschluss wird eine prototypische Implementierung von Introspected REST beschrieben.

3.1. Definition und Hauptbestandteile

Introspected REST ist ein Architekturstil, welcher fünf der sechs REST-Prinzipien unverändert beibehält, das Hypermedia-Prinzip aber durch das **Introspection-Prinzip** („Introspection as the Engine of Application State“) ersetzt. Dieses besagt, dass ein Client jederzeit die Möglichkeit haben soll, durch Introspection Informationen über die API und deren Ressourcen sowie über mögliche Zustandsübergänge und Aktionen (Hypermedia) zur Laufzeit abzufragen. Weiterhin fördert Introspected REST den Einsatz von Mediatypes, welche aus kleinen, zusammensetzbaren Einheiten, den **Microtypes**, bestehen.

Dieser Abschnitt beginnt mit der Definition grundlegender Begriffe des Datenmodels von Introspected REST. Danach folgt eine Beschreibung der Probleme von REST nach Vasilakis. Zum Schluss werden das Introspection-Prinzip sowie Microtypes näher erläutert.

3.1.1. Datenmodell

Vasilakis unterscheidet zwei Formen von Daten in Introspected REST: Nutzdaten und Metadaten. **Nutzdaten** beschreiben die eigentliche Repräsentation einer Ressource. Man könnte sie vage beschreiben als die Daten, welche den Endnutzer interessieren. **Metadaten** sind zusätzliche Informationen über eine Ressource (z.B. eine Beschreibung in Prosaform), eine Repräsentation (z.B. *JSON Schema* ([39; 40])) oder die Client-Server-Kommunikation (z.B. Pagination-Details). Hypermedia ist in diesem Modell eine spezielle Art der Metadaten. Weitere Arten von Metadaten sind u.a. Laufzeit-Metadaten, die sich von Request zu Request ändern können und bspw. Pagination-Informationen umfassen, strukturelle Metadaten, welche die Struktur der Nutzdaten beschreiben, und informationelle Metadaten, die Informationen für einen menschlichen Leser liefern [38, Abs. 9.1].

Nach Vasilakis existieren drei Formen von Hypermedia:

- LINKS sind ein Verweis auf eine Ressource, die in Beziehung zu der aktuellen Ressource steht

- ACTIONS sind Links, welche Informationen darüber enthalten, wie die referenzierte Ressource manipuliert werden kann
- FORMS dienen ebenfalls zur Manipulation von Ressourcen, sind aber semantisch äquivalent zu HTML Forms [38, Abs. 9.1.2]

Zur besseren Unterscheidung bezeichnet ein Link im Folgenden einen Verweis auf eine Ressource, welcher die Ressource nicht manipuliert, d.h. sicher ist, wenn man ihm folgt.

3.1.2. Probleme von REST

Introspected REST wurde entworfen, um die Probleme, die REST mit sich bringt, zu lösen, während die Vorteile von REST beibehalten und der Funktionsumfang nicht eingeschränkt werden sollte [38, Kap. 0, 2]. Vasilakis beschreibt folgende Probleme von REST, basierend auf seinen eigenen Erfahrungen und ohne Angabe von Belegen:

P1 ZU HOHE KOMPLEXITÄT: In gängigen REST-APIs wird Hypermedia in jedem Response des Servers mitgesendet. Dadurch steigt die Komplexität, mit der ein Client umgehen muss. Weiterhin ist es schwierig, dynamische Hypermedia-Elemente auf Serverseite zu implementieren und zu testen. Ein dynamisches Hypermediaelement ist bspw. ein Link, der nur für autorisierte Nutzer sichtbar ist [38, Abs. 8.2.1].

P2 MÖGLICHERWEISE NUTZLOSE INFORMATIONEN: Da im Vornherein nicht bekannt ist, welche Informationen ein Client benötigt, werden alle möglichen Hypermedia-Elemente zum Client gesendet. Dieser muss dadurch Informationen verarbeiten, die er nicht benötigt [38, Abs. 8.2.2]. Vor allem Clients, die Hypermedia überhaupt nicht verwenden, leiden unter der zusätzlichen Datenlast.

P3 EVOLVIERBARKEIT ZU LASTEN DER PERFORMANCE: Die unabhängige Weiterentwicklung von Server und Client wird durch Hypermedia möglich gemacht. Allerdings wächst durch viele Hypermedia-Elemente die Größe der Responses. Dadurch verlangsamt sich das Senden, Empfangen und Parsen [38, Abs. 8.2.3].

P4 KEIN HYPERMEDIA-CACHING: Obwohl sich Hypermediainformationen nur selten ändern, ist es nicht möglich, diese zu cachen, da sie in REST mit den Nutzdaten vermixt sind. Sind die Nutzdaten im Cache nicht mehr aktuell oder gar nicht erst cachebar, gilt dies auch für die Hypermedia-Elemente des jeweiligen Response [38, Abs. 8.2.4].

P5 SCHLECHTE HYPERMEDIA-EVOLVIERBARKEIT: Durch die Vermischung von Nutzdaten und Hypermedia ist es nicht möglich, die Hypermedia-Elemente unabhängig von der Ressource weiterzuentwickeln [38, Abs. 8.2.5].

P6 KEINE INKREMENTELLE ENTWICKLUNG: Wenn einer API, die noch über keine Hypermediaunterstützung verfügt, Hypermedia-Elemente hinzufügt werden, erfordert dies einen neuen Mediatype, da sonst die Semantik der API verändert werden würde. Hypermedia kann also nicht in abwärtskompatibler Art und Weise hinzugefügt werden [38, Abs. 8.2.6.1].

P7 KEINE KOMPOSITION: REST fördert nicht die Komposition verschiedener Teile einer API, die jeweils durch eine eigene Spezifikation beschrieben werden könnten. Vor allem die Mediatypes in HTTP sind große Monolithen, die alles an einem Ort beschrieben [38, Abs. 8.2.7.2].

3.1.3. Das Introspection-Prinzip

Das Hypermedia-Prinzip von REST schließt mit ein, dass ein Client kein vorheriges Wissen über die API benötigt mit Ausnahme der Einstiegs-URL sowie die Unterstützung der von der API verwendeten Mediatypes. Alle weiteren Informationen, die für die Interaktion mit dem Server benötigt werden, werden durch Hypermedia bereitgestellt [23]. Fielding hat, soweit es dem Autor bekannt, keine Aussage darüber getroffen, *wie* Hypermedia zum Client gelangen soll. Stattdessen wird nur das *Vorhandensein* von Hypermedia gefordert. Der heute übliche Ansatz ist, Hypermedia als Teil der Repräsentation zu senden. Hypermedia kann dabei einen Großteil der Nachricht ausmachen.

In Listing B.1 (siehe Anhang) ist der Payload eines HTTP-Response abgebildet. Der verwendete Mediatype ist die *Hypertext Application Language (HAL)* bzw. *application/hal+json* ([41]). HAL ist ein universeller Mediatype für REST mit Hypermedia-Unterstützung. Das Listing entstammt der Spezifikation und wird als repräsentatives Beispiel angenommen. In dem Response nimmt Hypermedia 15 von 30 Zeilen in Anspruch (alle `_links`-JSON-Propertys) – und dabei werden noch nicht einmal umfangreiche Informationen geliefert, wie Vasilakis in [38, Abs. 7.3] beschreibt. Für einen Client, welcher Hypermedia nicht verwendet, bedeutet die größere Nachricht eine längere Zeit bis zum vollständigen Empfangen und ein Mehraufwand beim Parsen. Die Integration von Hypermedia in jede Repräsentation, wie es bei REST-APIs heute üblich ist, trägt zu den Problemen **P1** bis **P5** bei.

Introspection bezeichnet die Fähigkeit eines Systems, Informationen über sich selbst abzufragen. Im Fall von Introspected REST soll ein Client Informationen über die API, deren Ressourcen, zulässige Aktionen sowie Metadaten durch Introspection-Requests erhalten [38, Abs. 9.3].

Das Introspection-Prinzip besagt, dass die Nutzdaten und Metadaten, vor allem Hypermedia, getrennt werden sollen. Nur Laufzeit-Metadaten dürfen mit den Nutzdaten kombiniert werden [38, Abs. 9.3.2]. Daraus folgt, dass Hypermedia nicht mehr Teil der Repräsentation einer Ressource sein darf. Stattdessen wird Hypermedia in einem

eigenen Request angefragt. Für Clients, die Hypermedia keine Beachtung schenken, verringert sich dadurch die Größe des ursprünglichen Response erheblich, denn den Introspection-Request müssen sie nicht senden. Hypermediaaffine Clients müssen dagegen einen zusätzlichen Request zum Server senden, was einen Mehraufwand bedeutet. Durch Multiplexing und Server Push dürften die Auswirkungen mit HTTP/2 allerdings vernachlässigbar sein.

Da Metadaten, mit Ausnahme der Laufzeit-Metadaten, sich nicht so schnell verändern wie die Nutzdaten, kann der Introspection-Response gecacht werden (siehe Problem **P4**). Auch wenn sich die Nutzdaten ändern, muss der Client keinen zusätzlichen Introspection-Request senden. Dadurch kann sich die Gesamtgröße empfangener Nachrichten sogar für hypermediaaffine Clients verringern. Durch Microtypes kann ein Client dem Server außerdem mitteilen, welche Art von Metadaten er empfangen möchte, was noch mehr Einsparungen bedeuten kann. Dies wird in Abschnitt 3.1.4 genauer erläutert.

Introspection setzt gewissermaßen „intelligente“ Clients voraus, weil diese die beiden Responses miteinander kombinieren müssen. Zum Beispiel müssen sie Nutzdaten in ein URI-Template des Introspection-Response einsetzen können.

3.1.4. Microtypes

Das Ziel von Microtypes ist es, einen Mediatype in mehrere unabhängige Komponenten aufzuteilen, welche jeweils einen Teil der Funktionalität einer API abbilden. Der Mediatype dient nur noch als Container für Microtypes und legt fest, wie Clients die benötigten Microtypes auswählen können. Microtypes weißen vier Eigenschaften auf. Sie sind:

- klein, d.h. sie haben eine einzige Verantwortung
- unabhängig, d.h. ein Client kann die Informationen eines Microtypes verstehen, ohne andere Microtypes kennen zu müssen
- wiederverwendbar, d.h. sie sollen in mehreren APIs eingesetzt werden können und es sollten Programmbibliotheken sowohl für die Client- als auch für die Serverentwicklung existieren
- konfigurierbar, z.B. über Parameter im HTTP-Accept-Header [42]

Content-Negotiation

Content-Negotiation ist der Prozess in HTTP, durch welchen die bestmögliche Repräsentation einer Ressource ausgewählt wird, abhängig von den Präferenzen des Clients und dem Angebot des Servers. Die Repräsentationen können sich u.a. in Kodierung, Sprache oder Mediatype unterscheiden. RFC 7231 definiert zwei Arten von Content-Negotiation: proaktive (servergetriebene) und reaktive (clientgetriebene). Für die **proaktive Content-Negotiation** sendet ein Client seine Präferenzen mit einem Request mit. Beispielsweise können dafür die HTTP-Accept-Header verwendet

werden. Der Server entscheidet anhand der bereitgestellten Informationen über die Repräsentation, welche er zurücksendet. Der *Apache HTTP Server*[1] verwendet z.B. den *httpd-Negotiation-Algorithmus* [43]. Bei der **reaktiven Content-Negotiation** sendet der Server eine Liste der möglichen Repräsentationen an den Client und dieser ruft die für ihn passende Repräsentation auf [17].

Client und Server sollen durch Content-Negotiation über die verwendeten Microtypes verhandeln können. Ein Client kann z.B. eine Präferenz für JSON als Datenformat angeben, *Problem Details* ([44]) als Format für Fehlermeldungen, Cursor-Pagination als Pagination-Mechanismus, *JSON-LD* ([45]) für semantische Anreicherungen und *GraphQL* als Query-Language für die Suche. Diese Spezifikationen müssten in einen entsprechenden Microtype verpackt werden. Content-Negotiation kann auch mit Introspection kombiniert werden. Beispielsweise kann ein Microtype existieren, der nur über die verfügbaren HTTP-Methoden der Ressource informiert. Weitere **Introspection-Microtypes** könnten *JSON Schema* oder *JSON Hyper-Schema* ([46]) einbinden, um Clients über das Schema der Ressource bzw. zusätzlich über vorhandene Hypermedia-Elemente zu unterrichten. Je nachdem, welche Informationen ein Client benötigt, kann er den jeweiligen Microtype auswählen. Damit ein Client weiß, welche Microtypes vorhanden sind, sollen Microtypes entdeckbar sein: Ein Server soll einen Client darüber informieren, welche Microtypes er anbietet.

Durch das Aufkommen vieler verschiedener mobiler Clients müssen große API-Anbieter mit einer Vielzahl von Benutzungsszenarien und Anforderungen an die API umgehen können. In der Folge sind Architekturmuster wie *Backends for Frontends* ([47]) oder neuen API-Stilen wie GraphQL [48] entstanden. Microtypes bilden den Lösungsansatz von Introspected REST für dieses Problem. Da Clients individuell die verwendeten Microtypes aushandeln können, ist eine viel bessere Anpassung an die individuellen Gegebenheiten möglich. API-Anbieter könnten selbst so weit gehen, dass Microtypes die Nutzdaten prägen. Zum Beispiel könnten eigene Microtypes existieren, um mehr oder weniger Details für Elemente einer Liste darzustellen. Content-Negotiation von Microtypes ist der Grund, weshalb Vasilakis Introspected REST nicht nur als Alternative des „klassischen“ RESTs darstellt, sondern auch als Alternative zu GraphQL.

Unabhängigkeit

Durch die Unabhängigkeit der Microtypes ist es möglich, einen Microtype durch einen anderen mit ähnlicher Funktionalität zu ersetzen, neue Microtypes hinzuzufügen und Microtypes zu verändern und zu erweitern, ohne dass der Mediatype oder andere Microtypes angepasst werden müssen. Anstatt also eine neue Version eines Mediatypes veröffentlichen zu müssen, wenn sich dessen Funktionsumfang ändert, kann der Server einen Client über einen neu vorhandenen Microtype informieren.

[1] Apache HTTP Server Project: https://httpd.apache.org/ (besucht am 15.09.2020)

Neue Clients können dann den neuen Microtype anfragen, verwenden aber die gleiche Version des Mediatypes wie noch nicht angepasste Clients. Werden Microtypes clientseitig als unabhängige Module implementiert, muss nur ein Modul geändert werden, um auf Änderungen an einem Microtype zu reagieren.

Wiederverwendbarkeit

Microtypes eröffnen das Potenzial für ein Ökosystem an API-Features, wie es Verborgh und Dumontier beschreiben. In einem solchen wird gleiche API-Funktionalität durch die gleiche Schnittstelle verkörpert. Zum Beispiel wäre die Schnittstelle für das Hochladen eines Fotos bei APIs verschiedener sozialer Netzwerke gleich. So könnten, wie im Human-Web, mit der Zeit wiedererkennbare Muster entstehen, sodass Entwicklerinnen und Entwickler sich nicht umgewöhnen müssen und der Einarbeitungsaufwand zurückgeht [49]. Für API-Anbieter würde sich der Designaufwand reduzieren, da sie auf bekannte Muster zurückgreifen können [50]. Gleichzeitig ermöglicht die Wiederverwendbarkeit von API-Features die Wiederverwendung von Code. Beispielsweise könnten Microtypes clientseitig in Bibliotheken umgesetzt werden, die über Paketmanager verteilt werden. Cliententwicklerinnen und -entwickler könnten so flexibel die benötigten Funktionen wählen, ohne an die Art und Weise eines bestimmten Mediatypes gebunden zu sein, während sie ohne großen Aufwand Bibliotheken miteinander komponieren. Clients würden auch universeller werden. So könnte ein Client für mehrere soziale Netzwerke verwendet werden mit minimalen Anpassungen an die jeweilige API.

3.2. Prototypische Implementierung

Für den Vergleich von Introspected REST mit anderen API-Stilen wird eine konkrete API benötigt. Es existieren allerdings bisher keine öffentlichen Implementierungen des Introspected-REST-Architekturstils. Deshalb wird im Folgenden eine Implementierung auf Basis von HTTP unter Verwendung von ASP.NET Core 3.1[2] vorgestellt. Der Quellcode für die Implementierung ist in `/Code/IntrospectedREST` zu finden. Wie bei REST benötigt ein Client zum Anfang eine Einstiegs-URL. Weiterhin muss er den Container-Mediatype sowie die für die API relevanten Microtypes verstehen.

3.2.1. Ein Container-Mediatype für Microtypes

Der Container-Mediatype dient als Hülle für die Microtypes, welche die eigentlichen Nutz- und Metadaten enthalten. Der im Folgenden verwendete, nicht-standardisierte

[2]Introduction to ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-3.1 (besucht am 02.09.2020)

Mediatype ist *application/vnd.microtype-container+json*. Er verfügt über zwei JSON-Propertys, welche das Datenmodell von Introspected REST abbilden: `data` und `meta`.

Die `data`-Property dient zum Transport der Nutzdaten und kann eine beliebige Form haben. Das heißt, es können Wahrheitswerte, Zahlen, Zeichenketten, null, Arrays und Objekte in der `data`-Property stehen. Die Struktur kann durch einen **Content-Microtype** beschrieben werden. Dieser wird im `Content-Type`-Header als `content`-Parameter angegeben. Content-Microtypes können z.B. Daten in Form von normalem JSON transportieren oder weitergehende Spezifikationen wie *JSON Home Documents* [51] nachbilden.

content/json: Kann beliebige Daten im JSON-Format (RFC 8259) transportieren. Es werden keine weiteren Aussagen über die Daten getroffen.

content/json-home: Stellt Informationen über die API bereit. Dieser Microtype folgt der Spezifikation für JSON-Home-Documents (*draft-nottingham-json-home-06*). Die JSON-Property `resources` enthält Links und ist nach Spezifikation nicht optional, muss aber für diesen Microtype ein leeres Objekt sein, da Links durch Introspection bereitgestellt werden.

`meta` ist der Platz für Laufzeit-Metadaten wie bspw. Pagination-Informationen. `meta` ist ein JSON-Objekt und verfügt über eine Property für jeden **Runtime-Microtype**, wobei der Schlüssel der Property dem Bezeichner des Microtypes entspricht. Der Wert der Property ist vom jeweiligen Microtype abhängig. Für jede Microtype-Kategorie sollte der verwendete Microtype im `Content-Type`-Header als Parameter angegeben werden. Listing B.2 (siehe Anhang) zeigt ein Beispiel eines Introspected-REST-Response. Der Wert des `Content-Type`-Headers für dieses Beispiel wäre:

```
Content-Type: application/vnd.microtype-container+json;content=json;
↪ pagination=offset-pagination
```

Fehlermeldungen können im `meta`-Teil angegeben werden. Dafür wird ein **Fehler-Microtype** verwendet. Es wird empfohlen, einen Standard-Microtype für diesen Fall festzulegen, auch wenn dieser nicht durch den Client verhandelt wurde. Ein geeigneter Standard sind die *Problem-Details für HTTP APIs*. Es ist möglich, dass die `data`-Property Nutzdaten enthält, obwohl Fehler vorhanden sind.

error/problem-details: Vermittelt zusätzliche Informationen über einen Fehler an den Client. Die Struktur des Microtypes entspricht dem *application/problem+json*-Mediatype (RFC 7807).

3.2.2. Content-Negotiation für Microtypes

Damit der Server weiß, welche Informationen der Client benötigt, verhandeln beide die verwendeten Microtypes. Wie in Abschnitt 3.1.4 beschrieben, gibt es in HTTP reaktive und proaktive Content-Negotiation. Die hier vorgestellte Implementierung verwendet die proaktive Variante mit `Accept`-Header. Einem HTTP-Header wurde Vorzug gegenüber Query-Parametern gegeben, weil die präferierten Microtypes Metainformationen über die Kommunikation und keine domänrelevanten Informationen darstellen. Es ist aufgrund der möglicherweise großen Anzahl an Permutationen von Microtypes nicht möglich, die ausgewählten Microtypes in dem Pfad der URL zu kodieren. Wie von Vasilakis vorgeschlagen [38, Abs. 10.2], werden die Microtypes als Parameter des Mediatypes angegeben.

```
Accept: application/vnd.microtype-container+json;
↪  pagination=offset-pagination;error=problem-details
```

Werden mehrere Microtypes für dieselbe Kategorie angegeben, wählt der Server den ersten unterstützten Microtype aus. Unterstützt der Server keinen der angegebenen Microtypes, antwortet er mit `406 Not Acceptable`.

Die Implementierung setzt zwar hauptsächlich auf proaktive Content-Negotiation, verbindet dies aber mit einer Übersicht unterstützter Microtypes. Diese Übersicht ist der Body des `406`-HTTP-Response, wenn keiner der angegebenen Microtypes akzeptiert wird. Somit ist eine Art der reaktiven Content-Negotiation möglich.

3.2.3. Introspection durch HTTP-OPTIONS

Introspection wird durch einen OPTIONS-Request ausgeführt, wie es von Vasilakis vorgeschlagen wurde [38, Abs. 10.4.1]. Clients können präferierte Introspection-Microtypes im `Accept`-Header des Requests angeben. Das Resultat ist in der `data`-Property des Container-Mediatypes enthalten. Im Gegensatz zu Nutzdaten, deren Content-Microtype im `content`-Parameter des `Content-Type`-Header angegeben wird, ist bei einem Introspection-Response der `content`-Parameter nicht gesetzt. Stattdessen wird der verwendete Introspection-Microtype im `introspection`-Parameter sichtbar gemacht. Das Problem an OPTIONS-Requests ist, dass keine Unterscheidung zwischen verschiedenen HTTP-Methoden für die gleiche Ressource getroffen werden kann. So muss im Introspection-Response sowohl `GET /notes` als auch `POST /notes` behandelt werden. Ein weiteres Problem ist, dass OPTIONS-Responses nach RFC 7231 nicht cachebar sind [17, Abs. 4.3.7]. Somit besteht das Problem **P4** bei diesem Ansatz weiterhin.

Ein möglicher Introspection-Microtype könnte die standardmäßige Funktionalität eines OPTIONS-Requests nachbilden und nur die zulässigen HTTP-Methoden für eine Ressource zurückgeben. Für Clients, die nur an den Hypermedia-Informationen

interessiert sind, wird ein Microtype, welcher nur die entsprechenden Hyperlinks zurückgibt, spezifiziert. Ein fortgeschrittenerer Microtype könnte JSON-Hyper-Schema verwenden, um das JSON-Schema des Response sowie zulässige Links und Actions an den Client zu übermitteln. Um das richtige Schema zu generieren, muss auch der `content`-Parameter des `Accept`-Headers berücksichtigt werden, denn das Schema ist abhängig vom verwendeten Content-Microtype.

introspection/methods-only: Gibt die zulässigen HTTP-Methoden im `Allow`-Header des Response zurück.

introspection/links-only: Gibt für jede Ressource die Links zu anderen Ressourcen zurück. Es gibt eine Property `links`, welche ein Array von Link Descriptor Objects nach *draft-handrews-json-schema-hyperschema-02* enthält.

introspection/json-hyper-schema: Enthält Metadaten über die durch die URI repräsentierte Ressource. Das Datenformat folgt der Spezifikation von JSON-Hyper-Schema (*draft-handrews-json-schema-hyperschema-02*). Zustandsübergänge und mögliche Aktionen können über die `links`-Property spezifiziert werden. Mögliche Konfigurationseinstellungen könnten sein: *include-self-link*, *include-type-schema*.

Weil ein Server mehrere Microtypes anbieten kann, soll er den Client über die vorhandenen Microtypes informieren, um eine reaktive Content-Negotiation zu ermöglichen. Dafür sendet er eine Übersicht über die vorhandenen Introspection-Microtypes an den Client. Weiterhin werden mögliche Content- und Runtime-Microtypes für die angegebene Ressource aufgelistet. Dieser **Übersichts-Microtype** ist ein guter Standardwert für Introspection-Responses und sollte verwendet werden, wenn kein `introspection`-Parameter im `Accept`-Header eines OPTIONS-Requests angegeben wurde. Ein Beispiel für einen Overview-Response findet sich in Listing B.3 (siehe Anhang).

introspection/overview: Enthält mögliche Content-Microtypes in der Property `content`, Runtime-Microtypes in der Property `runtime` und Introspection-Microtypes in der Property `introspection`. Jeder Microtype wird durch eine JSON-Property abgebildet, wobei der Schlüssel der Property dem Namen des Microtypes entspricht. Die Microtype-Property kann dadurch zusätzliche Informationen wie eine verbale Beschreibung, einen Link zur Dokumentation oder mögliche Konfigurationseinstellungen vermitteln. Runtime-Microtypes müssen über eine `category`-Property verfügen. Unterstützt eine Ressource mehrere HTTP-Methoden, können die Informationen pro Methode angegeben werden.

Eine Möglichkeit, um Microtypes, die in der gesamten API verfügbar sind, abzufragen, könnte sein, einen Asterisk als Pfad der Anfrage zu verwenden [17, Abs. 4.3.7]. Sinnvoll ist dies bspw. für Error-Microtypes. Ein Client könnte dadurch ein Vokabular

an beiderseits bekannten Microtypes aufbauen und eventuell ganz auf einen zusätzlichen OPTIONS-Request verzichten. Wie immer sollten Cliententwicklerinnen und -entwickler aber einen alternativen Weg über den OPTIONS-Request programmieren, falls der Server für eine Ressource doch nicht den gewünschten Microtype verwenden kann.

3.2.4. Implementierung in ASP.NET Core

Die Basis für eine Introspected-REST-API bildet eine ASP.NET Core Web-API. Zu der Request-Pipeline wird Middleware für Content-Negotiation und Introspection hinzugefügt. Die `ConnegMiddleware` überprüft, ob der Container-Mediatype vom Client akzeptiert wird. Weiterhin werden die akzeptierten Microtypes aus dem `Accept`-Header extrahiert und für später gespeichert.

Die `IntrospectionMiddleware` überprüft, ob der Request ein OPTIONS-Request ist. Falls ja, wird in den von der `ConnegMiddleware` extrahierten Microtypes nach einem Introspection-Microtype gesucht, der auch vom Server unterstützt wird. Die entsprechende Microtype-Implementierung ist dann dafür verantwortlich, den richtigen Response zurückzugeben. Als Eingabe erhalten die Introspection-Microtype-Implementierungen einen `ControllerActionDescriptor`, welcher Informationen über den durch die URL identifizierten Endpunkt enthält. Dafür muss die Middleware für Introspection nach der `EndpointRoutingMiddleware` von ASP.NET Core in der Request-Pipeline aufgerufen werden. Können sich Client und Server nicht auf einen Introspection-Microtype einigen oder ist im `Accept`-Header kein `introspection`-Parameter angegeben, wird die `OverviewIntrospection` ausgeführt.

`JsonHyperSchemaIntrospection` verwendet das `Returns`-Attribut an den Methoden der Controller, um den erwarteten Rückgabetypen herauszufinden. Für diesen Typen wird das JSON Schema erzeugt. Links können durch `Link`-Attribute an der entsprechenden Klasse definiert werden. Die `LinksOnlyIntrospection` extrahiert nur die Links von den in den `Returns`-Attributen angegebenen Klassen.

Ist der Request kein OPTIONS-Request, führt ASP.NET Core die Controller-Methode für den durch die URL identifizierten Endpunkt aus. Innerhalb dieser Methoden wird ein `RestResult` zurückgegeben, welches das `IActionResult`-Interface implementiert. Neben den Nutzdaten enthält ein `RestResult`-Objekt auch die vom Server unterstützten Runtime-Microtypes.

Im `RestResultExecutor` erfolgt die eigentliche Content-Negotiation. Die Nutzdaten werden an den verhandelten Content-Microtype übergeben und alle vom Client nicht akzeptierten Runtime-Microtypes werden entfernt. Danach wird der Response entsprechend der Spezifikation des Container-Mediatypes und der ausgewählten Microtypes als JSON-Objekt serialisiert und der Payload in den Response-Body geschrieben.

Die Implementierung ist ausgerichtet auf Erweiterbarkeit, sodass eigene Microtypes

einfach hinzugefügt werden können. Durch die Verwendung von ASP.NET Core werden viele Aufgaben wie bspw. Routing, Authentifizierung oder die Deserialisierung des Requests übernommen. Gerade für die Generierung des JSON Schemas werden viele Attribute verwendet, welche die Übersichtlichkeit des Codes verringern. Weiterhin lassen sich nur einige zur Übersetzungszeit feststehende Werte als Parameter an Attribute übergeben. [52, Abs. 22.2] In der zum Zeitpunkt der Abgabe existierenden Implementierung wurden noch nicht alle Funktionen umgesetzt. Beispielsweise fehlen die Konfigurationsmöglichkeiten für Microtypes und die Fehlerbehandlung mithilfe von Fehler-Microtypes.

3.3. Zusammenfassung

Introspected REST versucht, einige Probleme moderner REST-APIs zu lösen. Dazu baut es auf zwei Pfeilern auf: Introspection und Microtypes. Das Introspection-Prinzip besagt, dass Metadaten getrennt von den Nutzdaten ausgeliefert werden sollen, und schafft vor allem Vorteile für Clients, die kein Interesse an den Hypermedia-Elementen haben. Microtypes stellen kleine, unabhängige, wiederverwendbare und konfigurierbare Einheiten der API dar, welche eine bestimmte Funktionalität beschreiben. Durch Content-Negotiation kann ein Client die Funktionalität auswählen, die ihn interessiert.

Es wurde eine Implementierung basierend auf ASP.NET Core beschrieben. Dazu wurden auch einige Microtypes spezifiziert. Um eine Übersicht über alle verfügbaren Microtypes zu bekommen, existiert der Introspection-Microtype *introspection/overview*. Die Introspection wird durch einen OPTIONS-Request angefragt.

4. Kriterien für den Vergleich der API-Stile

In Abschnitt 3.1.2 wurden die Probleme von REST nach Vasilakis beschrieben. Das Ziel von Introspected REST ist es, diese Probleme zu beheben und trotzdem die Vorteile von REST beizubehalten [38, Kap. 2]. Die genannten Probleme wirken sich auf die folgenden Faktoren aus:

- Probleme **P3** und **P4** auf Performance
- Probleme **P5**, **P6** und **P7** auf Evolvierbarkeit
- Probleme **P1** und **P2** auf Komplexität

In der vorliegenden Arbeit soll Introspected REST mit alternativen API-Stilen hinsichtlich dieser drei Faktoren verglichen werden. In diesem Kapitel werden Performance, Evolvierbarkeit und Komplexität definiert und es werden Wege aufgezeigt, mit denen sich die genannten Kriterien erfassen lassen. Dazu wird in Abschnitt 4.1 ein Experiment zur Messung der Performance vorgestellt. Abschnitt 4.2 beschäftigt sich mit dem Vergleich der Evolvierbarkeit, wenn sich neue, konkrete Anforderungen an die APIs ergeben. Und in Abschnitt 4.3 wird die Komplexität auf Benutzbarkeit zurückgeführt und die Heuristische Evaluation für die Bewertung gewählt.

4.1. Performance

Performance ist eine wichtige nicht-funktionale Anforderung an viele Softwaresysteme [53; 54] und spielt auch für Web-APIs eine große Rolle. Eine Web-API wird zwar nicht von Endanwenderinnen und -anwendern direkt verwendet, allerdings benutzen diese Anwendungen, welche mit der API kommunizieren. So trägt eine hohe Performance der API zu einem besseren Nutzererlebnis bei und kann die Entscheidung für oder wider einen Web-API-Stil beeinflussen.

4.1.1. Performancemetriken für Web-APIs aus Clientperspektive

Performancemetriken liefern Informationen über die Performance eines Systems. Dabei verbinden verschiedene Stakeholderinnen und Stakeholder verschiedene Erwartungen und somit Metriken mit dem Begriff „Performance“ [55, S. 23]. Dem Anbieter einer Web-API ist eine gute Skalierung der Server bei effizienter Ressourcenausnutzung wichtig, sodass die durchschnittliche Serverauslastung für diesen eine wesentliche Metrik ist. Für Cliententwicklerinnen und -entwickler ist die vom Client wahrgenommene Performance von Bedeutung, welche durch die durchschnittliche Antwortzeit quantifizierbar ist. Letztgenannte Perspektive muss auch von Anbietern von Public-APIs berücksichtigt werden, da, wie in Abschnitt 2.1 beschrieben, das

Ziel einer Public-API die weite Verbreitung ist. Eine gute Performance kann dazu beitragen, dass sich Entwicklerinnen und Entwickler für die API eines Anbieters entscheiden und gegen die eines Konkurrenten.

In dieser Arbeit wird die Performance aus der Perspektive eines API-Clients analysiert, denn es wird angenommen, dass die Auswirkungen eines API-Stils auf die vom Client wahrgenommene Performance größer sind als die Auswirkungen auf serverinterne Metriken. Die wichtigste Metrik ist dabei die Antwortzeit. Die Antwortzeit (Response Time, RT) ist die Zeit, welche vom Senden des ersten Bytes des Requests bis zum Empfangen des letzten Bytes des Response vergeht. Sie ist indirekt proportional zur Performance Ω.

$$\Omega \sim \frac{1}{RT} \tag{4.1}$$

Die Antwortzeit setzt sich zusammen aus der Zeit für die Übertragung der Anfrage (Message Transmission Time of Request, MTT_{Req}), der Zeit für die Verarbeitung der Anfrage auf dem Server (Server Time, ST) und der Zeit für die vollständige Übertragung der Antwort zum Client (Message Transmission Time of Response, MTT_{Resp}) [56, Abs. 6.1].

$$RT = MTT_{Req} + ST + MTT_{Resp} \tag{4.2}$$

Die Übertragungszeit einer Nachricht wiederum setzt sich zusammen aus der Latenz des Netzwerks (Latency of Network, Lat_{Net}) sowie der Nachrichtengröße (Message Length, Len_{Msg}) geteilt durch die Datenübertragungsrate des Netzwerks (Data Transfer Rate, DTR) [57, S. 83]. Latenz und Datenübertragungsrate sind dabei Eigenschaften des Netzwerks und unabhängig von dem verwendeten API-Stil.

$$MTT_{Msg} = Lat_{Net} + \frac{Len_{Msg}}{DTR} \tag{4.3}$$

Die Serverzeit ist abhängig von dem gewählten API-Stil, allerdings vor allem auch von dessen Implementierung: Je nach Framework oder Bibliothek, welche auf dem Server eingesetzt wird, kann die Zeit für die Bearbeitung einer Anfrage variieren. Ein Server kann die Server-Timing-API [58] nutzen, um einem Client Informationen über die für interne Abläufe benötigten Zeiten zu übermitteln.

Für den Vergleich der Performance verschiedener API-Stile aus Perspektive eines API-Clients wird die Antwortzeit RT ermittelt. Weiterhin erfolgt die Messung der Nachrichtengrößen von Request Len_{Req} und Response Len_{Resp}, welche Rückschlüsse auf die Serverzeit zulassen. Um die Ursachen einer hohen oder niedrigen Antwortzeit besser bewerten zu können, wird die Anzahl benötigter Requests berücksichtigt.

4.1.2. Experiment zur Erfassung der Metriken

Für den Vergleich der Performance von Introspected REST und alternativen API-Stilen wird ein szenariobasiertes Experiment durchgeführt. Ein Experiment dient zur quantitativen empirischen Untersuchung [59]. Es werden bestehende Hypothesen überprüft [60, S. 42], Beziehungen quantifiziert und Zusammenhänge offengelegt [61, S. 9].

Unabhängige Variablen in einem Experiment sind die Einflussfaktoren, welche kontrolliert und verändert werden können. **Abhängige Variablen** werden durch die unabhängigen Variablen beeinflusst. Ein **Faktor** bezeichnet die unabhängigen Variablen, deren Effekte in dem Experiment untersucht werden sollen. Ein **Treatment** ist ein bestimmter Wert eines Faktors [61, S. 74f.]. In einem Experiment wird das Verhalten eines Systems untersucht, wenn ein Faktor in der Versuchsanordnung verändert wird, während alle anderen unabhängigen Variablen konstant gehalten werden [61, S. 11].

Von den empirisch-quantitativen Methoden zur Datenerhebung eignet sich ein Experiment am besten für den Vergleich der Performance. Ein solches ermöglicht es, die unabhängigen Variablen, welche Auswirkungen auf die Performance haben, zu kontrollieren. So können die Resultate auf eine Änderung eines einzigen Faktors, in diesem Fall des verwendeten API-Stils, zurückgeführt werden. Andere Einflüsse wie die Latenz und Datenübertragungsrate eines Netzwerks sowie verwendete Technologien können weitestgehend konstant gehalten werden.

Ziel und Kontext

Das Ziel des Experiments ist es, die vier API-Stile Introspected REST, REST, GraphQL und gRPC hinsichtlich ihrer Performance aus Perspektive eines API-Clients zu untersuchen, um einen Vergleich zwischen den API-Stilen zu ermöglichen. Aufgrund des begrenzten zeitlichen Umfangs dieser Arbeit wird das Experiment in einem künstlichen Kontext durchgeführt. Die Untersuchung findet anhand eines einfachen, ausgedachten Szenarios statt. Die abhängigen Variable in diesem Experiment ist die Performance der API aus Clientperspektive. Unabhängige Variablen sind die Latenz des Netzwerks, die Datenübertragungsrate und die Performance von Client und Server. Der einzige Faktor in diesem Experiment ist der verwendete API-Stil. Treatments sind Introspected REST, REST, GraphQL und gRPC.

Für das Experiment wird die folgende Hypothese formuliert:

- Nullhypothese H_0: Die ermittelte Performance der API ist bei allen vier untersuchten API-Stilen gleich.
- Alternativhypothese H_1 (ungerichtet, unspezifisch): Die ermittelte Performance der vier untersuchten API-Stile unterscheidet sich.

Experimentdesign

Um einen möglichst realitätsnahen Performancevergleich zu ermöglichen, wird ein szenariobasierter Ansatz gewählt: Ein Client muss dafür vorgegebene Aufgaben erledigen. Dieser Ansatz ermöglicht einen Vergleich trotz Unterschiede bei den Interaktionsmustern der API-Stile. Um eine Aufgabe zu erledigen, benötigt ein Client ggf. mehrere Requests. Die Erhebung der Metriken erfolgt für die gesamte Kommunikation zwischen Client und Server während der Abarbeitung einer Aufgabe.

Das untersuchte Szenario ist *NeverNote*, eine Anwendung zum gemeinsamen Verwalten von Notizen. Die NeverNote-API bietet die Möglichkeit, neue Notizen anzulegen sowie bestehende zu ändern und zu löschen. Wird eine Notiz von einer Person bearbeitet, die nicht der Urheberin oder Urheber dieser Notiz ist, wird sie in die Liste der Bearbeiter dieser Notiz aufgenommen. Es können weiterhin Informationen über die registrierten Autorinnen und Autoren abgerufen werden und neue Autorinnen und Autoren können sich registrieren.

Für den Performancevergleich müssen Clients folgende Aufgaben bewältigen:

T1 Zeige die ID und den Titel aller Notizen sowie den Benutzernamen der jeweiligen Urheberinnen und Urheber an.

T2 Zeige den Titel, Inhalt und das Erstellungsdatum der ersten Notiz aus **T1** an.

T3 Zeige die Schlagwörter aller Notizen an, die von den Bearbeiterinnen und Bearbeitern der ersten Notiz aus **T1** bearbeitet wurden.

T4 Füge das neue Schlagwort „Neu“ zur ersten Notiz aus **T1** hinzu.

T5 Lösche alle Notizen, die über das Schlagwort „Mathematik“ verfügen.

Für jede dieser Aufgaben werden vier Experimente durchgeführt. Im ersten Experiment gibt es nur einen Client, welcher jede Aufgabe 250 Mal bewältigt. Im zweiten Experiment arbeiten vier Clients zeitgleich, um jede Aufgabe jeweils 100 Mal zu bearbeiten. Es wurden vier Clients für das zweite Experiment gewählt, da dies der Anzahl der logischen Kerne des Prozessors des Versuchsgeräts entspricht. Beide Experimente werden jeweils zwei Mal mit unterschiedlichen Bandbreiten durchgeführt. Der Grund, die Bandbreite zu begrenzen, liegt an der Auflösung der JMeter-Zeiterfassung von einer Millisekunde. Folglich muss die Kommunikation zwischen Client und Server länger als eine Millisekunde dauern, um sinnvolle Antwortzeiten erfassen zu können. Bei vier API-Stilen werden insgesamt $4 * 2 * 2 = 16$ Messungen durchgeführt.

Instrumentation

Zur Erfassung der Messwerte wird *Apache JMeter 5.3*[1] verwendet. JMeter ist eine Anwendung für Performancemessungen und Lasttests von Client-Server-Anwendungen. Diese ermöglicht die Messung der Antwortzeit (*Elapsed Time* [62]) sowie der Nachrichtengrößen. Die Bandbreitenbegrenzung erfolgt mithilfe von *NetLimiter 4*[2].

Ein **Sampler** in JMeter sendet einen Request zum Server und erfasst dabei die Messwerte. Ein **Thread** führt eine Liste von Samplern und anderen Komponenten nacheinander aus. Dabei arbeitet er unabhängig von anderen Threads. Andere Komponentenarten sind u.a. Listener, welche die erfassten Messwerte auswerten und darstellen, sowie Pre- und Post-Prozessoren, welche eine Aktion vor bzw. nach dem Sampler-Request ausführen. Eine **Thread-Gruppe** ist ein Zusammenschluss von Threads, welche parallel ausgeführt werden [63].

Für das Experiment werden in JMeter für jeden API-Stil zwei Thread-Gruppe angelegt: Eine mit einem Thread und eine andere mit vier Threads. Für Introspected REST, REST und GraphQL wird der Standard-HTTP-Sampler ohne TLS-Verschlüsselung verwendet, welcher eine HTTP/1.1-Verbindung aufbaut. Für gRPC-Requests wird das *JMeter gRPC Plugin*[3] installiert, welches auf Java und der gRPC-Integration in dieser Programmiersprache basiert. gRPC benutzt HTTP/2 mit TLS.

Für jeden API-Stil wird eine Web-API auf Basis von ASP.NET Core 3.1 entwickelt. Alle APIs verfügen über den gleichen Funktionsumfang, wobei bei der Umsetzung die Charakteristika des jeweiligen Stils beachtet wurden. Um die Wiederholbarkeit zu gewährleisten, werden niemals Änderungen der Daten ausgeführt, folglich haben Aufgaben **T4** und **T5** keine Seiteneffekte. Für alle API-Stile wird zu Beginn der Messung ein Request ausgeführt, welcher nicht mit in die Betrachtung einfließt, um die TCP-Verbindung zum Server zu etablieren.

Die Introspected-REST-API verwendet die in Kapitel 3 vorgestellte Bibliothek. Als Introspection-Microtype wird *introspection/links-only* verwendet. Für die GraphQL-API wird *GraphQL.NET*[4] verwendet, für die gRPC-API *gRPC for .NET*[5], welches offiziell von Microsoft unterstützt wird.

Für die REST-API wurde der Mediatype *JSON:API* bzw. *application/vnd.api+json* [64] gewählt. Für die Implementierung wurde das *JSON API .Net Core*-Framework[6] verwendet, welches auf ASP.NET Core aufbaut. JSON:API ist für eine grobe Unterteilung der API-Ressourcen und eine kleine Anzahl an Requests gedacht. Deshalb können die referenzierten Ressourcen, bspw. die Urheberin oder der Urheber einer Notiz, in den Response eingeschlossen werden (sog. *Compound Documents*). Somit

[1] Apache JMeter™: https://jmeter.apache.org/ (besucht am 06.09.2020)
[2] NetLimiter: https://www.netlimiter.com/ (besucht am 11.09.2020)
[3] JMeter gRPC Plugin (Repository): https://github.com/zalopay-oss/jmeter-grpc-plugin (besucht am 06.09.2020)
[4] GraphQL.NET: http://graphql-dotnet.github.io/ (besucht am 06.09.2020)
[5] gRPC for .NET (Repository): https://github.com/grpc/grpc-dotnet

könnten alle benötigten Daten mit nur einem Request geholt werden [64]. Diese Funktionalität steht aber beim Großteil der verbreiteten REST-Mediatypes nicht zur Verfügung. Um die Vergleichbarkeit mit dem Introspected-REST-Stil zu erhöhen, wird diese Funktionalität nicht verwendet, sodass mehrere Requests nötig sind, um eine Notiz (erster Request) und deren Urheberin oder Urheber (zweiter Request) abzufragen.

Die Introspected-REST- und REST-API geben für alle Ressourcen einen `Cache-Control`-Header mit `max-age=60` zurück. In JMeter werden gecachte Requests nicht noch einmal gesendet. Allerdings werden dann auch die Post-Prozessoren nicht ausgeführt. Deshalb wird ein Cache nur für einzelne Aufgaben und nicht im gesamten JMeter-Client eingesetzt, sodass einige Requests trotzdem erneut gesendet werden. Es werden keine Conditional-Requests unterstützt und die Responses enthalten weder einen `ETag`- noch einen `Last-Modified`-Header. OPTIONS-Requests werden nicht gecacht. Der Cache wird nach jeder Iteration geleert.

Alle APIs greifen auf die gleichen Daten zu, welche über die `NotesRepository`- und `AuthorsRepository`-Klasse abgefragt werden können. Die Daten liegen im Arbeitsspeicher. Um eine neue Notiz hinzufügen oder eine bestehende ändern zu können, muss ein `Authorization`-Header für Basic-Authentifizierung gesendet werden. Der dort angegebene Benutzername muss einer Autorin oder einem Autor zugeordnet sein. Die Autorin oder der Autor werden dann als Urheberin oder Urheber einer neuen Notiz bzw. Bearbeiterin oder Bearbeiter einer bestehenden Notiz hinzugefügt. Um die Requests besser identifizierbar zu machen, wird ein `Request-No`-Header mit jedem Request gesendet.

Das Experiment wird lokal auf einem Surface Pro 4 (8GB RAM, Intel® Core™ Prozessor i5-6300U) mit Windows 10 Pro (Version 2004) eingesetzt. JMeter und die API-Server laufen auf dem gleichen Gerät und kommunizieren über die Loopback-Schnittstelle. Dadurch ist keine (messbare) Latenz vorhanden und die Datenübertragungsrate ist nur abhängig von der begrenzten Bandbreite. Der Code für die erstellen APIs ist in den beigelegten Dateien unter `/Code/NeverNote` zu finden. Der JMeter-Testplan wurde unter `/Code/JMeter/performance.jmx` hinzugefügt.

4.1.3. Validität und Übertragbarkeit der Ergebnisse

Das ausgewählte Szenario ist einfach gehalten und deshalb nicht repräsentativ für eine API der realen Welt. Es werden Aufgaben gestellt, welche sich vor allem um das Abfragen und einfache Manipulieren von Daten drehen, da sich diese mit allen API-Stilen umsetzen lassen. Die untersuchten API-Stile werden in der Praxis aber meist für unterschiedliche Zwecke eingesetzt. So können mit REST komplexe Workflows umgesetzt werden, während sich dies mit gRPC und GraphQL schwieriger gestaltet.

[6] JSON API .Net Core (Repository): https://github.com/json-api-dotnet/JsonApiDotNetCore/ (besucht am 06.09.2020)

gRPC wurde vor allem für Interprozesskommunikation entwickelt und nicht für Public-APIs. GraphQL dient vor allem für Datenabfrage und -manipulation und ist deshalb für CRUD-artige APIs prädestiniert. Die gestellten Aufgaben sind der kleinste gemeinsame Nenner der untersuchten API-Stile.

Introspected-REST- und REST-APIs tauschen mehr Daten mit dem Client aus, da zusätzlich Hypermedia-Elemente und eventuell weitere Metadaten gesendet werden. Dies führt zu größeren Nachrichten verglichen zu GraphQL und gRPC. Da dies aber in den Charakteristika der API-Stile begründet ist, beeinflusst diese Tatsache die Untersuchung nicht negativ.

Mit ASP.NET Core wird zwar die gleiche Plattform für den API-Server verwendet, allerdings werden unterschiedliche Bibliotheken für die API-Implementierungen eingesetzt. Nur diejenige für gRPC wird offiziell von Microsoft unterstützt.

Für gRPC wird HTTP/2 verwendet, während alle anderen API-Stile über HTTP/1.1 kommunizieren. Zwar existiert ein *HTTP2 Plugin for JMeter*[7], dieses hat aber bei den ersten Versuchen zu Problemen geführt. HTTP/2 bietet eine bessere Kompression der HTTP-Header sowie Unterstützung für Multiplexing und Streaming, wodurch die gRPC-Anwendung einen Vorteil erhält.

Client und Server laufen auf dem gleichen Gerät und nicht in einer isolierten Umgebung.

In JMeter werden die Sampler innerhalb eines Threads sequentiell ausgeführt. Gerade bei Introspected REST und REST, aber auch bei gRPC ist es in der Realität möglich, viele der Requests parallel auszuführen, sodass die vom Benutzer wahrgenommene Zeit eigentlich weit unter den in diesem Experiment erfassten Messwerten liegen würde. Die parallele Ausführung wäre entweder über mehrere TCP-Verbindung oder durch Multiplexing in HTTP/2 möglich.

Die Auflösung der Zeiterfassung in JMeter beträgt nur eine Millisekunde. Bei kleinen Messergebnissen für die Antwortzeit entstehen also große Messungenauigkeiten. JMeter rundet die Ergebnisse immer auf ganze Millisekunden ab.

Für das Experiment wurden nur zwei unterschiedliche Bandbreiten untersucht, deren Werte zwar wohlbegründet, aber dennoch beliebig sind. Es ist dadurch nicht sichergestellt, dass die Ergebnisse auf reale Netzwerkanwendungen übertragbar sind.

Introspected REST und REST sind Architekturstile, während GraphQL und gRPC viel konkreter definiert sind. Die Ergebnisse für Introspected REST und REST sind also nicht ohne weiteres übertragbar. Beispielsweise hängt viel von dem verwendeten Mediatype ab.

[7] HTTP2 Plugin for JMeter (Repository): https://github.com/Blazemeter/jmeter-http2-plugin (besucht am 06.09.2020)

4.1.4. Verwandte Arbeiten

Da bisher noch keine wissenschaftliche Literatur zu Introspected REST existiert, bildet die vorliegende Arbeit den ersten Performancevergleich von Introspected REST mit alternativen API-Stilen. Es existieren jedoch Arbeiten, welche die Performance von REST und GraphQL untersuchen. Es konnte keine Literatur gefunden werden, welche sich explizit mit der Performance von gRPC-APIs beschäftigt oder diese empirisch untersucht.

Landeiro und Azevedo [65] haben in einer Fallstudie zwei Prototypen für eine GraphQL-API auf Basis einer bestehenden REST-API entwickelt und konnten teils erhebliche Performanceverbesserungen feststellen. Eine ähnliche Untersuchung haben Seabra et al. [66] angestellt. Dafür haben sie drei Anwendungen jeweils als REST- und als GraphQL-API implementiert und konnten auch hier einen Performancevorteil von GraphQL gegenüber REST ausmachen. Cederlund [67] hat in einer studentischen Arbeit GraphQL und Falcor, ein anderes Framework für deklarative Datenabfragen, verglichen und die Ergebnisse auch einer REST-API gegenübergestellt. Eine weitere studentische Arbeit wurde von Eizinger [68] angefertigt, in welcher er REST und GraphQL anhand mehrerer Kriterien, darunter auch Performance, vergleicht. Auch Gustavsson und Stenlund [69] vergleichen in ihrer Masterarbeit die Performance einer REST- und einer GraphQL-API.

4.2. Evolvierbarkeit

Wie alle Softwareprodukte muss sich auch eine Web-API an Veränderungen anpassen können. Nach der initialen Veröffentlichung können Fehler entdeckt werden, sich die Anforderungen ändern, sich technologische Änderung vollziehen oder neue Benutzungsszenarien erschlossen werden. Deshalb ist es wichtig, die API so reibungslos wie möglich anpassen zu können – sowohl server- als auch clientseitig.

4.2.1. Definition und Einteilung für Web-APIs

Breivold et al. definieren **Softwareevolvierbarkeit** als „the ability of a software system to adapt in response to changes in its environment, requirements and technologies that may have impact on the software system in terms of software structural and/or functional enhancements, while still taking the architectural integrity into consideration.“ [70] Rowe et al. betrachten Evolvierbarkeit vor allem aus einer Kostenperspektive: Jene soll die Anpassung des Systems an veränderte Anforderungen mit kleinstmöglichen Kosten ermöglichen. Dabei grenzen sie Evolution von der Wartung und Instandhaltung des Systems ab. Während bei der Wartung die Anforderungen gleich bleiben, werden durch Evolution zuvor nicht spezifizierte Anforderungen umgesetzt [71].

Eine API bildet einen Vertrag zwischen Client und Server. Wenn sich dieser Vertrag ändert, kann man zwischen abwärtskompatiblen und abwärtsinkompatiblen Änderungen unterscheiden. Bei **abwärtskompatiblen Änderungen** (engl.: *non-breaking Changes*) funktioniert die Kommunikation zwischen Client und Server weiterhin wie vor der Änderung, es wird also keine Anpassung des Clients erzwungen [4, S. 215]. Das bedeutet aber nur, dass bspw. kein Fehler auftritt; die Semantik von Request und Response können sich trotzdem ändern. Außerdem sind auch bei abwärtskompatiblen Änderungen eventuell Anpassungen des Clients nötig, um eine neue Funktionalität nutzen zu können. Meistens ist das Hinzufügen neuer Elemente in der API abwärtskompatibel [72]. **Abwärtsinkompatible Änderungen** (engl: *breaking Changes*) erfordern eine Anpassung des Clients, damit die Kommunikation mit dem Server wieder funktioniert [4, S. 215]. Wird der Client nicht angepasst, treten bspw. Protokollfehler (etwa `404 Not Found`), Fehler bei der Validierung des Requests auf dem Server oder Fehler während der Verarbeitung des Response auf. Das Ändern oder Löschen bestehender Elemente der API ist meistens abwärtsinkompatibel [72]. Lauret listet in [4, Kap. 9] einige abwärtsinkompatible Änderungen und deren Konsequenzen auf.

4.2.2. Durchführung eines Evolvierbarkeit-Vergleichs für API-Stile

Um die Evolvierbarkeit von Introspected REST und alternativer API-Stile zu vergleichen, werden Änderungen an APIs und deren Auswirkungen auf einen Client untersucht. Für eine Cliententwicklerin oder einen -entwickler ist vor allem relevant, *ob* eine Änderung Anpassungen am Client erfordert, d.h. ob diese abwärtskompatibel oder -inkompatibel ist. Falls eine Anpassung erforderlich ist, müssen sie wissen, wie umfangreich die Änderungen des Clients ausfallen.

Für den Vergleich wird das Szenario der NeverNote-API aus Abschnitt 4.1.2 wieder aufgegriffen. Folgende Änderungen werden untersucht:

C1 Zur Registrierung neuer Autorinnen und Autoren wird ein neues erforderliches Feld *Geschlecht* hinzugefügt.

C2 Zur Registrierung neuer Autorinnen und Autoren wird ein neues optionales Feld *Nationalität* hinzugefügt.

C3 Das Feld *Email* der Autorinnen und Autoren wird entfernt.

C4 Der Begriff „Autor“ wird durch „Benutzer“ ersetzt.

C5 Zu den Notizen wird ein neues Feld *veröffentlicht* vom Typ Boolean hinzugefügt. Wird eine neue Notiz erstellt, ist der Wert des Felds `false`.

C6 Es wird ein neuer Arbeitsschritt eingeführt, um eine Notiz zu veröffentlichen.

C7 Eine Notiz kann nur veröffentlicht werden, wenn der Wert des Felds *veröffentlicht* `false` ist.

C8 Der Zugriff auf fremde Notizen, die noch nicht veröffentlicht wurden, erfordert Autorisierung.

C9 Bei der Abfrage von Notizen wird die Möglichkeit, nach Werten der Felder zu filtern, eingeführt.

Bei der Untersuchung einer Änderung wird davon ausgegangen, dass die vorherige Änderung schon im Client implementiert wurde. Weiterhin wird angenommen, dass Best-Practices eingehalten wurden. Dazu zählt, dass Objekte in GraphQL nur wiederverwendet werden, wenn die semantische Übereinstimmung gegeben ist [30, S. 53ff.] und dass in gRPC explizite Nachrichtentypen für jede Methode verwendet werden [73, 07:29–08:25]. Eine dritte Annahme ist die der „idealen Clients“. Das bedeutet, dass die Clients auf übliche Fehlermeldungen des jeweiligen API-Stils reagieren können. Für die Introspected-REST-, REST- und GraphQL-APIs können ideale Clients auch so weit wie möglich Hypermedia und/oder Introspection nutzen, um sich automatisch anzupassen und bspw. das Userinterface basierend auf diesen Informationen zu rendern.

4.2.3. Validität und Übertragbarkeit der Ergebnisse

Die Änderungen der Anforderungen wurden durch den Autor alleine ausgewählt und könnten deshalb einer Voreingenommenheit unterliegen. Die gewählten Aufgaben sind konkrete Ausprägungen einer allgemeinen Änderung. Beispielsweise treffen die Ergebnisse für Aufgabe **C1** auf jede Änderung zu, bei welcher ein neues erforderliches Feld zu einer Aktion hinzugefügt wird. Deshalb sind die Ergebnisse gut übertragbar, obwohl die Untersuchung in einem sehr konkreten Kontext stattfindet.

4.2.4. Verwandte Arbeiten

Das Thema *Evolvierbarkeit* ist vor allem im Bereich der REST-APIs verbreitet, angefangen bei Fielding und Taylor [16] über Block et al. [74] bis zu Amundsen [75]. Lauret [4] gibt Tipps, wie man APIs erstellt, welche abwärtskompatible Anpassungen ermöglichen. Für GraphQL und gRPC existieren Hinweise zur Vermeidung abwärtsinkompatibler Änderungen sowie zur Versionierung bei Giroux [30] und Michela [73]. Es konnte aber keine Arbeit gefunden werden, welche die Evolvierbarkeit der API-Stile empirisch untersucht, geschweige denn die Evolvierbarkeit dieser miteinander vergleicht.

4.3. Komplexität

Komplexität ist eine wichtige Eigenschaft von Softwaresystemen, da sie Einfluss nimmt auf die Entwicklungszeit und Wartbarkeit [76]. In der Literatur wird der Begriff meist sehr oberflächlich und ohne konkrete Definition verwendet. Vor allem bedeutet Komplexität auch immer etwas anderes, je nach Blickwinkel, aus welchem man ein System betrachtet: Für Entwicklerinnen und Entwickler ist bspw. die Komplexität des Quellcodes von großer Bedeutung, während für Endanwenderinnen und Endanwender die Komplexität der Benutzeroberfläche, d.h. die Benutzbarkeit/Benutzerfreundlichkeit (engl.: *Usability*), wichtig ist.

4.3.1. Arten der Komplexität bei Web-APIs

Mitra unterscheidet zwei Arten von Komplexität bei Web-APIs: Systemkomplexität und Interfacekomplexität [77, 04:08–04:25]. Die **Systemkomplexität** ist umso höher, je mehr Komponenten und Konnektoren in einem System vorhanden sind und je häufiger Kommunikation zwischen diesen stattfindet. Die **Interfacekomplexität** bezieht sich auf die Benutzbarkeit einer API.

Für Clients einer API, welche HTTP verwendet, bleibt die Systemkomplexität, die das Netzwerk mit sich bringt, meist verborgen, da zusätzliche Komponenten wie Caches und Proxys transparent sind: Unabhängig davon, wie viele Vermittler zwischen Client und Server stehen, sieht es für den Client immer so aus, als würde er direkt mit dem Server kommunizieren. Aus Clientperspektive kann die Systemkomplexität aber z.B. dadurch erhöht werden, dass eine zusätzliche Abhängigkeit wie ein SDK für die API eingeführt wird.

In Abschnitt 2.1 wurden Kosteneinsparungen als Ziel von Private-APIs und eine weite Verbreitung als Ziel von Public-APIs genannt. Zur Umsetzung beider Ziele ist ein wichtiger Bestandteil, die Einarbeitungszeit und den Aufwand der Cliententwicklerinnen- und entwickler zu reduzieren [78]. Neben einer guten Dokumentation, einem guten Support und niedrigen Einstiegshürden spielt die Komplexität einer API dafür eine herausragende Rolle[8] [4, S. 9]. Für Cliententwicklerinnen und -entwickler ist der Begriff Komplexität mit Interfacekomplexität – und damit der Benutzbarkeit der API – gleichzusetzen, da die Systemkomplexität durch die API verborgen wird. Deshalb wird im Vergleich von Introspected REST mit alternativen API-Stilen die Interfacekomplexität bzw. Benutzbarkeit betrachtet.

[8] Um Kosten zu sparen, müssen Anbieter allerdings auch abwägen, wie viel Komplexität auf Seiten des Servers und wie viel auf Seiten des Clients behandelt werden soll, da nach Teslers *Gesetz über die Erhaltung der Komplexität* ein bestimmter Teil der Komplexität nicht vermindert, sondern nur verschoben werden kann [79, S. 136].

4.3.2. Bewertung der Benutzbarkeit einer Web-API

Zur Bewertung der Benutzbarkeit existieren verschiedene Strategien. Zu diesen zählen:

- Theoriebasierte Bewertung, z.B. Heuristische Evaluation [80], Cognitive Dimensions Framework [81; 82], Cognitive Walkthrough [83; 84]
- Befragungen, z.B. System Usability Scale [85], Fokusgruppen, Interviews
- Empirische Tests, z.B. Think-Aloud-Technik [86], Usability-Tests [87], Eyetracking

Heuristische Evaluationen werden dabei oft verwendet, obwohl sie oft in der Kritik stehen, da sie Zeit und Kosten sparen im Vergleich zu empirischen Tests. Die Kritik richtet sich vor allem gegen die fehlende Validität und Vollständigkeit der identifizierten Probleme. [88; 89] Empirische Tests mit echten Benutzerinnen und Benutzern werden bei Untersuchungen der Mensch-Computer-Interaktion als das Nonplusultra angesehen [78].

Aufgrund des zeitlich begrenzten Umfangs dieser Arbeit wird trotz der geringeren Effektivität eine Heuristische Evaluation von Introspected REST und alternativen API-Stilen durchgeführt. Dafür werden Heuristiken aus verschiedenen Quellen zusammengetragen, die für Web-APIs relevant sind. Es existieren nicht viele solcher Heuristiken für Web-APIs selbst. Für APIs von Bibliotheken und Frameworks wurden allerdings einige Untersuchungen angestellt, etwa bei de Souza und Bentolila [90], Zibran [91] oder Scheller und Kühn [92]. Ebenfalls existieren Richtlinien, welche den Erfahrungen der Autorinnen und Autoren entsprungen sind, bspw. bei Bloch [93]. Als weitere Quelle dienen allgemeine Usability-Prinzipien und -Heuristiken, wie sie Lidwell et al. [94] oder Nielsen [95] aufgestellt haben.

In Abschnitt 5.3 werden die Heuristiken zusammen mit ihrer Anwendung auf die unterschiedlichen API-Stile vorgestellt. Es erfolgt eine Bewertung, wie gut einzelnen API-Stilen den jeweiligen Heuristiken entsprechen. Dafür wird folgende Skala verwendet: `-` (entspricht Heuristik überhaupt nicht), `+` (entspricht Heuristik bedingt), `++` (entspricht Heuristik zum Großteil), `+++` (entspricht Heuristik voll und ganz). Es wird keine Gewichtung der Heuristiken angegeben. Leserinnen und Leser sollten jedoch berücksichtigen, dass sich nicht alle Heuristiken gleich stark auf die Benutzerfreundlichkeit auswirken.

4.3.3. Validität und Übertragbarkeit der Ergebnisse

Wie bei einer Heuristischen Evaluation allgemein kann nicht sichergestellt werden, dass die Liste der verwendeten Heuristiken vollständig und jede Heuristik auch geeignet ist. Da der Autor kein Experte im Bereich der Benutzerfreundlichkeit ist, besteht vor allem die Gefahr, dass Heuristiken falsch, z.B. außerhalb des eigentlichen Kontexts, angewendet werden. Dadurch könnte eine negative Bewertung stattfinden, obwohl die Benutzerfreundlichkeit nicht negativ beeinflusst wird. Bei der Auswahl

der Heuristiken hat der Autor auf sein Empfinden vertraut, ob eine Heuristik auf Web-APIs übertragbar ist. Die Auswahl ist folglich subjektiv und könnte von Voreingenommenheit geprägt sein. Ein weiteres Problem ist, dass die Evaluation nur durch einen einzigen Gutachter stattfindet und so der subjektive Eindruck noch verstärkt wird. Um die Bewertung möglichst absolut und nicht abhängig vom Ergebnis anderer API-Stile zu machen, wurden alle Heuristiken auf einmal für einen API-Stil untersucht, nicht jeder API-Stil für jede Heuristik. Trotzdem kann nicht ausgeschlossen werden, dass das Ergebnis der Bewertung eines API-Stils durch die vorherige Bewertung eines anderen beeinflusst wird.

Durch die fehlende Gewichtung der Heuristiken wird es nicht ermöglicht, die Benutzbarkeit nach Durchführung der Bewertung quantitativ zu beurteilen. Das Ergebnis des Vergleichs muss also unter einer von Fall zu Fall gewählten Gewichtung betrachtet werden, welche von Faktoren wie der Vertrautheit der Entwicklerinnen und Entwickler mit einem API-Stil oder dem Bedarf an Flexibilität abhängt. Die relative Bewertung der API-Stile bezüglich einer Heuristik stuft der Autor aber als allgemein übertragbar ein.

4.3.4. Verwandte Arbeiten

Es existieren viele Arbeiten, welche sich mit der Benutzbarkeit von APIs für Programmbibliotheken beschäftigen. Einige Autorinnen und Autoren wurden in Abschnitt 4.3.2 genannt. Mosqueira-Rey et al. [96] geben einen Überblick über die vorhandene Literatur im Bereich API-Usability, stellen allgemeine Heuristiken heraus und ordnen diese in eine Taxonomie ein. Robillard [97] untersucht, wie eine API besser erlernt werden kann. Als Beispiel einer Usability-Studie für APIs sei die Untersuchung von Piccioni et al. [98] genannt. Ein Repertoire an Veröffentlichungen hält die Website *API Usability*[9] bereit. Ebenfalls mit Komplexität, wenn auch nicht von APIs, beschäftigt sich Lilienthal in ihrer Dissertation [99]. Darin führt sie die Komplexität von Softwarearchitektur auf Phänomene der kognitiven Psychologie zurück.

4.4. Zusammenfassung

Performance wird durch Metriken beschrieben, welche Informationen über das Systemverhalten liefern aus verschiedenen Blickwinkeln liefern. In dieser Arbeit wird die Perspektive eines API-Clients eingenommen. Eine relevante Metrik ist hier vor allem die Antwortzeit RT. Für den Vergleich wird ein Experiment durchgeführt, in welchem API-Clients Aufgaben lösen und dafür bestimmte Requests an den Server senden müssen. Dafür wurde für jeden untersuchten API-Stil eine API auf Basis von

[9] API Usability – Publications: https://sites.google.com/site/apiusability/resources/user-studies (besucht am 14.09.2020)

ASP.NET Core erstellt. Apache JMeter dient als API-Client und zur Erfassung der Messwerte.

Evolvierbarkeit bezeichnet die Eigenschaft einer API, sich an veränderte Bedingungen anpassen zu können. Dazu können abwärtskompatible und -inkompatible Änderungen vollzogen werden. Für den (analogischen) Vergleich der Evolvierbarkeit von Introspected REST und alternativen API-Stilen werden Änderungen an einer existierenden API simuliert und die Auswirkungen auf den Client beschrieben.

Komplexität bedeutet aus Sicht von API-Nutzerinnen und -Nutzern vor allem *Benutzbarkeit*. Die Benutzbarkeit der API-Stile wird in dieser Arbeit durch eine Heuristische Evaluation bewertet. Dafür kommen Heuristiken für APIs von Bibliotheken, aber auch allgemeine Usability-Prinzipien zum Einsatz.

5. Vergleich der API-Stile

In diesem Kapitel werden die Ergebnisse des Vergleichs von Introspected REST mit REST, GraphQL und gRPC vorgestellt. Dazu werden erst die Ergebnisse des Performance-Experiments beschrieben. Danach wird die Evolvierbarkeit analysiert, indem geänderte Anforderungen simuliert werden. Im dritten Abschnitt erfolgt eine Heuristische Evaluation der API-Stile.

5.1. Performance

Die 16 Messungen wurden wie in Abschnitt 4.1.2 beschrieben durchgeführt. Für jeden einzelnen Request wurden, neben anderen Messwerten, die Antwortzeit sowie die Größen der gesendeten und empfangenen Nachrichten durch JMeter erfasst (siehe Dateien `/Messdaten/*/results.jtl`). Die Bezeichnungen der Requests wurde mit einer Nummer versehen, sodass sich die Requests leichter zu den Aufgaben zuordnen lassen. Die Requests für eine Aufgabe wurden im folgenden Schritt gebündelt, d.h. die Antwortzeiten und Nachrichtengrößen wurden addiert, sodass insgesamt #Aufgaben $*$ #Iterationen Stichproben für jede Messung vorlagen. Für jede Aufgabe wurde nun das arithmetische Mittel der Messwerte über den Iterationen ermittelt. Die Ergebnisse werden in den folgenden Abschnitten beschrieben. Grafische Darstellungen der Ergebnisse befinden sich in Anhang A.

5.1.1. Ermittlung von Requestanzahl und Nachrichtengrößen

In Tabelle 5.1 wurde die Anzahl der für jede Aufgabe benötigten Requests je API-Stil notiert. In GraphQL können vier der fünf Aufgaben durch einen einzigen Request erledigt werden. Nur für die letzte Aufgabe ist erst eine Abfrage nötig, um danach Mutationen ausführen zu können. Da die gRPC-API grobkörniger gestaltet ist als die REST- und Introspected-REST-API, werden insgesamt weniger Requests benötigt. Beispielsweise werden in Aufgabe **T3** alle Keywords mit der Notiz zurückgegeben, während bei REST ein zusätzlicher Request zu `/notes/id/keywords` nötig ist. Mit der Introspected-REST-API werden ungefähr doppelt so viele Requests wie mit der REST-API benötigt, da zusätzlich ein eigener Request für die Introspection ausgeführt wird. Die Requests, welche bei Introspected REST und REST gecacht wurden (Zahl in Klammern), konnten beantwortet werden, ohne dass eine Verbindung mit dem Server aufgebaut werden musste. Durch die Verwendung von JSON:API als Mediatype und Bibliothek für die REST-API werden keine Responses in Aufgabe **T1** gecacht, da auf dieselbe Autorin oder denselben Autor durch unterschiedliche URLs verwiesen wird. Beispielsweise geben `/notes/5/creator` und `/notes/6/creator`

eine Repräsentation derselben Ressource zurück, trotzdem müssen beide Requests ausgeführt werden.

	Int. REST	REST	GraphQL	gRPC
Aufgabe 1	26 (+6)	11	1	5
Aufgabe 2	1	1	1	1
Aufgabe 3	30 (+5)	14 (+5)	1	3
Aufgabe 4	3	2	1	1
Aufgabe 5	11	6	2	5

Tabelle 5.1.: Anzahl der benötigten (+gecachten) Requests je Aufgabe

In Tabelle 5.2 sind die Summen der Größen von Requests bzw. Responses für jede Aufgabe zusammengefasst. Die Größe der Responses ist bei GraphQL geringer als bei den anderen API-Stilen. Bei gRPC stechen vor allem die sehr kleinen Requests hervor, doch sind die Responses im Vergleich zu GraphQL sehr groß, da keine exakte Auswahl der Daten stattfindet. Beispielsweise werden bei Aufgabe **T5** Id, Titel, Inhalt und Schlüsselwörter aller Notizen mit Schlagwort „Mathematik“ zurückgegeben, obwohl nur die ID benötigt wird. Die Nachrichtengrößen von Introspected REST und REST liegen in der gleichen Größenordnung. Eine Ausnahme bildet hier Aufgabe **T1**, da jede Notiz noch einmal vollständig heruntergeladen werden muss, um den Wert der `creator.id`-Property zu lesen und das URI-Template zu füllen, sodass dann die Urheberin oder den Urheber abgerufen werden können. Durch ein API-Design, welches besser auf dieses Szenario zugeschnitten ist, hätte diese Situation vermieden werden können. Die Gesamtanzahl der Requests von Introspected REST ist in etwa doppelt so groß wie die von REST. Die Gesamtzahl gesendeter Bytes bei Introspected REST ist folglich höher, da zu jedem Request auch eine Request-Line und die HTTP-Header gehören. Selbiges gilt im Übrigen auch für Responses: Die Status-Line und die HTTP-Header machen pro Response um die 200 Bytes aus. Im Vergleich zu GraphQL und gRPC wird bei Introspected REST und REST zusätzlich Hypermedia zum Client gesendet.

	Int. REST	REST	GraphQL	gRPC
Aufgabe 1	5666/16996	2017/9901	298/961	16/5325
Aufgabe 2	214/612	173/784	359/340	4/438
Aufgabe 3	5987/14132	2595/15453	440/928	12/6595
Aufgabe 4	805/1296	543/1137	448/249	9/443
Aufgabe 5	2817/4841	1325/3095	772/436	28/1290

Tabelle 5.2.: Gesamtgröße der gesendeten/empfangenen Nachrichten in Bytes

5.1.2. Ermittlung der Antwortzeiten

Tabelle 5.3 zeigt die Antwortzeiten für die Aufgaben je API-Stil, wenn ein bzw. vier Clients simultan arbeiten. Die Messung wurde mit einer begrenzten Bandbreite von 34KB/s von Client zu Server und 58KB/s von Server zu Client durchgeführt.

Für Aufgabe **T2** wurde bei jedem API-Stil nur ein einziger Request benötigt. Die gRPC-API ist in diesem Fall am schnellsten, gefolgt von Introspected REST und GraphQL. Sind die Aufgaben aber komplexer, punktet die GraphQL-API, da oft nur ein einziger Request benötigt wird und zusätzlich nur ausgewählte Daten zurückgegeben werden. Werden Requests von vier Clients gleichzeitig gesendet, steigt die durchschnittliche Antwortzeit um viel mehr als das Vierfache.

	Int. REST		REST		GraphQL		gRPC	
Clients	1	4	1	4	1	4	1	4
Aufgabe 1	215	1204	159	1045	6	51	60	327
Aufgabe 2	8	48	11	85	7	52	3	45
Aufgabe 3	194	1123	257	1422	9	60	62	247
Aufgabe 4	17	115	9	176	10	64	10	52
Aufgabe 5	53	419	60	404	14	116	24	264

Tabelle 5.3.: Antwortzeiten in ms mit Bandbreite 34KB/s Up-, 58KB/s Download

In Tabelle 5.4 sind die Ergebnisse der gleichen Messung mit einer begrenzten Bandbreite von 100KB/s zwischen Client zu Server für beide Richtungen dargestellt. Für Aufgabe **T3** mit einem Client lässt sich nunmehr kein nennenswerter Unterschied in den Antwortzeiten von gRPC, GraphQL und Introspected REST feststellen. Auch bei dieser Bandbreite erhöht sich mit drei zusätzlichen Clients die Antwortzeit um viel mehr als das Vierfache.

	Int. REST		REST		GraphQL		gRPC	
Clients	1	4	1	4	1	4	1	4
Aufgabe 1	86	549	88	600	2	8	35	182
Aufgabe 2	3	23	7	55	2	8	3	26
Aufgabe 3	76	540	151	830	2	12	28	146
Aufgabe 4	7	63	5	101	3	7	4	32
Aufgabe 5	22	214	34	242	5	22	14	147

Tabelle 5.4.: Antwortzeiten in ms mit Bandbreite 100KB/s Up-, 100KB/s Download

5.2. Evolvierbarkeit

In der folgenden Tabelle wird angegeben, ob eine Anpassung der Clients für die Änderungen aus Abschnitt 4.2.2 zwangsläufig ist, d.h. ob die Änderung abwärtskompatibel ist oder nicht. Gilt die gleiche Aussage für Introspected REST und REST, wurden beide API-Stile in einer Zeile zusammengefasst.

API-Stil	abwärts-komp.	Bemerkungen
C1 (Neues erforderliches Feld in Request):		
(Intro.) REST	Ja	Ein idealer Client kann durch Hypermedia automatische Anpassungen bei Änderungen der Objekte vornehmen und neue Felder anzeigen (vgl. [75, S. 211ff.]). Sind Clients nicht in dieser Weise programmiert, schlägt die Kommunikation mit dem Server bei der Registrierung neuer Autorinnen oder Autoren fehl (`400 Bad Request`).
GraphQL	Ja	Durch Introspection ist es für einen idealen GraphQL-Client möglich, automatisch neue Felder anzuzeigen. Sind Clients nicht in dieser Weise programmiert, schlägt die Validierung des Querys fehl.
gRPC	Nein	Damit Clients die neu hinzugefügten Felder sehen, benötigen sie die aktualisierte Servicedefinition und müssen die Änderungen umsetzen. Bis dahin schlägt die Kommunikation mit dem Server fehl, da das Feld erforderlich, der Wert aber immer `null` ist.
C2 (Neues optionales Feld in Request):		
(Intro.) REST	Ja	Wie bei Änderung **C1**, allerdings schlägt die Kommunikation von suboptimalen Clients mit dem Server bei der Registrierung neuer Autorinnen oder Autoren nicht fehl, da das Feld optional ist.
GraphQL	Ja	Wie bei Änderung **C1**, allerdings schlägt die Validierung des Querys von suboptimalen Clients nicht fehl, da das Feld optional ist.
gRPC	Ja	Clients haben zwar keine Möglichkeit, das neue Feld ohne die aktualisierte Servicedefinition zu sehen, allerdings können sie weiterhin Requests für die Registrierung senden, ohne dass ein Fehler auftritt.

API-Stil	abwärts-komp.	Bemerkungen
C3 (Feld aus Response entfernt):		
(Intro.) REST	Nein	Wenn die Anzeige des Clients auf diesem Feld beruht, ist das Entfernen des Felds nicht abwärtskompatibel. Es ist unrealistisch, dass ein Client diese Änderung automatisch integriert.
GraphQL	Nein	Ist das entfernte Feld im Selection-Set enthalten, schlägt die Validierung des Querys fehl.
gRPC	Nein	Dem entfernten Feld wird der Standardwert, im Fall der E-Mail also ein leerer String, zugewiesen. Wird das Feld im Client verwendet, treten Fehler auf.
C4 (Begriff umbenannt):		
(Intro.) REST	Ja	Bleiben die Bezeichnungen aller Link-Relations gleich, ist keine Anpassung der Clients erforderlich. In der NeverNote-API wird die Bezeichnung „Autor“ nur in den URLs verwendet; die relevanten Link-Relations sind *creator* bzw. *contributors*. URLs können ohne Anpassungen geändert werden, solange Clients Hypermedia verwenden.
GraphQL	Nein	Die Ersetzung des Begriffs „Autor“ hat die Umbenennung einiger Felder zur Folge. Sind diese Felder im Selection-Set enthalten, schlägt die Validierung des Querys fehl.
gRPC	Nein	Da gRPC-Methoden umbenannt werden, tritt ein `Unimplemented`-Fehler auf. Würden nur Nachrichten oder Felder umbenannt werden, kann die Kommunikation zwischen Client und Server trotz der Änderung stattfinden, solange die Struktur der Nachrichten gleich bleibt.
C5 (Feld zu Response hinzugefügt):		
(Intro.) REST	Ja	Alle Clients sollten Felder, die sie nicht kennen, ignorieren [75, S. 223f].
GraphQL	Ja	Das neue Feld wird nicht vom Server zurückgegeben, da es nicht im Selection-Set enthalten ist.
gRPC	Ja	Das neue Feld wird nicht deserialisiert, der Client spürt also keine Änderung.

API-Stil	abwärts-komp.	Bemerkungen
C6 (Arbeitsschritt hinzugefügt):		
(Intro.) REST	Ja	Alle bestehenden Clients funktionieren weiterhin. Suboptimale Clients müssen allerdings angepasst werden, um mit ihnen eine Notiz veröffentlichen zu können. Ein idealer Client kann dynamisch die zulässigen Aktionen anzeigen und einen „Veröffentlichen“-Button neben einer Notiz rendern.
GraphQL	Ja	Bestehende Clients funktionieren weiterhin. Um allerdings eine Notiz veröffentlichen zu können, sind Anpassungen am Client erforderlich. Dynamisches Rendern der zulässigen Aktionen ist schwierig, da man mit Mutations keine Hierarchie bilden kann. Es ist also nicht bekannt, zu welchen Daten eine Aktion gehört.
gRPC	Ja	Bestehende Clients funktionieren weiterhin. Um allerdings eine Notiz veröffentlichen zu können, muss die neue Servicedefinition geholt und implementiert werden.
C7 (Arbeitsschritt beschränkt):		
(Intro.) REST	Ja	Der Server kontrolliert die zulässigen Zustandsübergänge. Wenn eine Notiz schon veröffentlicht wurde, sendet er das entsprechende Hypermedia-Element nicht. Ein Client kann diesen Status im Userinterface umsetzen.
GraphQL	Nein	Das Schema ermöglicht es nur durch das Typsystem, Vorbedingungen für Mutations festzulegen. Dass ein Feld einen bestimmten Wert haben muss kann nicht deklariert werden. Die Mutation ist also immer noch syntaktisch korrekt, der Server gibt allerdings einen Fehler zurück.
gRPC	Nein	Der Server gibt einen Fehler zurück, wenn die Vorbedingungen verletzt werden. Um den Client anzupassen, muss die Dokumentation befragt und die Businesslogik im Client hinterlegt werden.

API-Stil	abwärts-komp.	Bemerkungen
C8 (Autorisierung erforderlich):		
(Intro.) REST	Ja	Versucht eine Benutzerin oder ein Benutzer auf eine nicht veröffentlichte Notiz zuzugreifen, gibt der Server `403 Forbidden` oder `401 Unauthorized`. Beides sind bekannte HTTP-Status, folglich kann ein idealer Client damit umgehen. Fehlen die Anmeldeinformationen, könnte der Client ein entsprechendes Formular einblenden und den Request wiederholen.
GraphQL	Nein	Verfügt eine Benutzerin oder ein Benutzer nicht über ausreichende Berechtigungen, wird entweder `null` oder eine Fehlermeldung für das entsprechende Feld zurückgegeben. Beide Fälle erfordern, dass die Logik im Client angepasst wird. Es gibt keine Standardfehlercodes für GraphQL.
gRPC	Ja	Der Server antwortet mit Fehler `UNAUTHENTICATED` oder `PERMISSION_DENIED`. Ein idealer Client kann automatisch auf diese Fehler reagieren.
C9 (Neue API-Funktionalität hinzugefügt):		
Int. REST	Ja	Der Server stellt einen neuen Microtype für das Filtern der Ergebnisse bereit. Clients, die den neuen Microtype unterstützen, können ihn mit dem Server aushandeln. Clients ohne Unterstützung funktionieren weiterhin.
REST	Ja	Wird die neue Funktionalität in einem Mediatype umgesetzt (wie bei JSON:API), muss eine neue Version dessen geschaffen und vom Client ausgehandelt werden. Wird die neue Funktionalität durch Hypermedia ausgedrückt, kann ein idealer Client sich bei manchen Hypermediaelementen automatisch anpassen.
GraphQL	Ja	Die neue Funktionalität wird ausgeprägt als optionales Argument eines Felds. Bestehende Clients funktionieren folglich weiterhin. Falls die möglichen Werte des Arguments im Typsystem kodiert werden, kann ein idealer Client das neue Feld automatisch rendern.
gRPC	Ja	Bestehende Clients verwenden die neue Funktionalität zwar nicht, die Kommunikation mit dem Server funktioniert aber weiterhin.

Zählt man alle Jas zusammen, schneiden Introspected REST und REST sowie GraphQL und gRPC gleich gut ab, wobei Introspected REST/REST bei den untersuchten Änderungen eine bessere Evolvierbarkeit aufweisen als die beiden anderen API-Stile.

5.3. Komplexität

In Abschnitt 4.3 wurde die Komplexität als Benutzerfreundlichkeit betrachtet. Es wurden Methoden zur Bewertung der Benutzerfreundlichkeit genannt und die Heuristische Evaluation für den Vergleich von Introspected REST mit alternativen API-Stilen ausgewählt. In diesem Abschnitt werden Heuristiken aufgestellt und die API-Stile hinsichtlich dieser bewertet.

Myers und Stylos beschreiben sechs Eigenschaften von APIs, welche die Benutzbarkeit beeinflussen: Erlernbarkeit, Produktivität, Fehlervermeidung, Einfachheit, Konsistenz und Übereinstimmung mit mentalen Modellen der Benutzerinnen und Benutzer [78]. Die genannten Eigenschaften können sich gegenseitig beeinflussen.

In den folgenden Abschnitten werden Heuristiken für die Eigenschaften Erlernbarkeit, Fehlervermeidung, Einfachheit und Konsistenz vorgestellt. Für jeden API-Stil wird daraufhin bewertet, wie gut er die jeweilige Heuristik umsetzt. Die verwendete Skala wurde in Abschnitt 4.3 festgelegt. Für Produktivität sowie die Übereinstimmung der API mit mentalen Modellen konnten keine geeigneten Heuristiken gefunden werden oder diese wurden aus Platzgründen weggelassen.

5.3.1. Heuristiken für die Erlernbarkeit

Um die API für neue Benutzerinnen und Benutzer einfach erlernbar zu machen, sollten Hindernisse abgebaut, Feedback gegeben und die Möglichkeiten der API aufgezeigt werden. Weiterhin spielt die Konsistenz für die Erlernbarkeit eine Rolle.

Einfacher Einstieg

Der Benutzung der API sollten wenige Hindernisse im Weg stehen. Die Zeit, bis eine Benutzerin oder ein Benutzer den ersten Request senden kann, sollte möglichst klein gehalten werden. Der Einstiegspunkt der API sollte über die Möglichkeiten, welche die API bietet, informieren und dabei auf Ablenkung verzichten [94, S. 80].

Für die Benutzung von Introspected REST, REST und GraphQL wird nur ein HTTP-Client benötigt. Ist dieser vorhanden, kann bei den beiden ressourcenorientierten API-Stilen sofort ein Request an den Einstiegsendpunkt gesendet werden. Welche Informationen die API im Response preisgibt, ist allerdings stark vom verwendeten Media-/Microtype abhängig. Eine REST-API könnte z.B. ein Home-Dokument ([51])

verwendet werden, welches sowohl allgemeine Informationen über die API als auch Links zu Ressourcen und möglichen Aktionen bereitstellt. Da aber REST und Introspected REST Architekturstile und keine Spezifikation sind, ist die Verwendung nicht vorgeschrieben. Wird HTTP als Protokoll verwendet, kann die API sicher, d.h. ohne unerwünschte Seiteneffekte, mit den *sicheren Methoden* GET, HEAD und OPTIONS erkundet werden [17, Abs. 4.2.1].

GraphQL-Services werden durch ein Schema beschrieben, welches durch Tools wie *GraphiQL*[1] oder den *Altair GraphQL Client*[2] automatisch heruntergeladen wird. Derartige Tools zeigen die Felder an, welche die API anbietet. Damit kann die API sehr einfach erkundet werden. Durch die Konvention, dass Querys seiteneffektfrei sind, ermöglicht auch GraphQL eine sichere Erkundung.

Für gRPC ist die Generierung des Client-Stubs ein notwendiger Schritt. Die Generierung erfolgt entweder händisch mit dem Protocol Buffer Compiler *protoc* oder integriert in Build-Management-Tools wie *Apache Maven*[3] oder *MSBuild*[4]. Beide Wege erfordern das Tätigwerden der Benutzerinnen und Benutzer vor der ersten eigentlichen Benutzung. Die Einstiegshürde wird dadurch erhöht. Ist der Client-Stub aber generiert, können die möglichen Operationen, die der Server ausführen kann, eingesehen werden. Viele Entwicklungsumgebungen bieten darüber hinaus auch Autovervollständigung an.

	Int. REST	REST	GraphQL	gRPC
Einfacher Einstieg	++	++	+++	+

Sichtbarkeit von Aktionen

Ein System sollte klar mögliche Aktionen sowie die Konsequenzen dieser darstellen. Menschen sind besser darin, eine Option von mehreren zu wählen, als sich ohne Gedankenstütze an die richtige Option zu erinnern [94, S. 178, 250; 87, S. 129ff.].

Mögliche Aktionen sichtbar zu machen ist die Aufgabe von Hypermedia in Introspected REST und REST. Je nach verwendeten Micro- und Mediatypes können Hypermedia-Elemente auch Hinweise auf die Konsequenzen, d.h. die Semantik eines Zustandsübergangs, geben. Durch die klar definierte Semantik der HTTP-Methoden, mit Ausnahme von POST, kann man die Auswirkungen eines Requests vorhersagen. Allerdings haben Benutzerinnen und Benutzer meist keine Informationen darüber, welche Struktur ein Response aufweist (kein Schema).

In GraphQL und gRPC werden mögliche Aktionen durch das Schema bzw. die Schnittstellendefinition sichtbar. GraphQL bietet darüber hinaus eine Unterscheidung

[1] GraphQL IDE (Repository): https://github.com/graphql/graphiql (besucht am 07.09.2020)
[2] Altair GraphQL Client: https://altair.sirmuel.design/ (besucht am 07.09.2020)
[3] Apache Maven Project: https://maven.apache.org/ (besucht am 07.09.2020)
[4] Microsoft MSBuild: https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild (besucht am 07.09.2020)

zwischen Querys und Mutations. Die Konsequenzen eines Querys sind, dass die Daten in der im Selection-Set angegebenen Form zurückgegeben werden, ohne Seiteneffekte. Mutations verfügen demgegenüber keine klare Semantik. Diese muss durch den Namen der Mutation deutlich gemacht werden. In gRPC ist es nur durch die Benennung möglich, die Semantik zum Ausdruck zu bringen. Deshalb sind die Konsequenzen der Aktionen nicht so gut sichtbar.

	Int. REST	REST	GraphQL	gRPC
Sichtbarkeit von Aktionen	++	++	++	+

Aussagekräftige Fehlermeldungen

Fehlermeldungen sollten in normaler Sprache formuliert werden, das Problem präzise beschreiben und Lösungsansätze vorschlagen [87, S. 142ff.].

Introspected REST und REST verwenden HTTP-Statuscodes, um Fehler zu markieren. Diese sind vielen Webentwicklerinnen und -entwicklern bekannt. Weiterhin existiert mit *Problem Details for HTTP APIs* ([44]) ein Standard, um detailliertere menschenlesbare Informationen über einen Fehler in den Response zu integrieren. Introspected REST ermöglicht es zusätzlich, den verwendeten Fehler-Microtype zu verhandeln.

GraphQL-Fehlermeldungen zielen darauf ab, von Menschen gelesen zu werden: „Every error must contain an entry with the key message with a string description of the error intended for the developer as a guide to understand and correct the error.“ [28] gRPC verfügt über wohldefinierte Statuscodes[5] für verschiedene Situationen. Zusätzlich können menschenlesbare Beschreibungen der Fehler angefügt werden. Kein API-Stil erzwingt die Erstellung aussagekräftiger Fehlermeldungen.

	Int. REST	REST	GraphQL	gRPC
Aussagekräftige Fehlermeldungen	+	+	++	+

Hilfreiche Dokumentation

Zwar ist es besser, wenn Benutzerinnen und Benutzer nicht auf die Dokumentation zurückgreifen müssen. Trotzdem sollte diese vorhanden sein, falls Hilfe benötigt wird [87, S. 148ff.].

Für Introspected-REST- und REST-APIs besteht die Dokumentation hauptsächlich aus der Beschreibung der Micro- bzw. Mediatypes und der Zustandsübergänge. Fielding formuliert es wie folgt:

[5] Status codes and their use in gRPC: https://grpc.github.io/grpc/core/md_doc_statuscodes.html (besucht am 10.09.2020)

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. [23]

Im Gegensatz zu anderen HTTP-APIs müssen keine HTTP-Methoden oder URLs beschrieben werden, da diese Informationen in Hypermedia-Elementen eingebettet sind. Die verfügbaren Mediatypes für REST-APIs bleiben allerdings meist hinter der Erwartung zurück, umfangreiche Informationen über einen Request, den der Client senden kann, zu liefern. Beispielsweise ist es in *HAL* [41] nicht möglich, die HTTP-Methode einer Aktion anzugeben. Auch gibt es keine Möglichkeit, den Payload eines POST- oder PUT-Requests zu beschreiben. Dafür werden durch *Compact URIs* (CURIEs) [100] Links zur Dokumentation einer Link-Relation zur Verfügung gestellt, sodass Benutzerinnen und Benutzer auf diese Weise an die benötigten Informationen kommen. Mediatypes wie *Siren*[6] verfügen über die Möglichkeit, menschenlesbare Beschreibungen der Relations zu Links und Actions hinzuzufügen. *Profiles* ermöglichen es, erweiterte Informationen über die Semantik einer API zur Verfügung zu stellen. Diese zielen aber vor allem auf die Maschine-zu-Maschine-Kommunikation ab [101]. Im Allgemeinen gibt es keinen einheitlichen Standard oder Tooling für die Dokumentation von REST-APIs.

Ähnlich wie bei REST verhält es sich auch bei Introspected REST. Zusätzlich dokumentiert werden müssen hier die einzelnen Microtypes und deren Konfigurationsmöglichkeiten. Die einzelnen Optionen können zwar durch Introspection abgefragt werden, doch muss eine Möglichkeit geschaffen werden, die Semantik zu definieren.

In GraphQL werden Kommentare in der Schemadefinition bei vielen Tools als Dokumentation zur Verfügung gestellt, sodass die einzelnen Felder und Argumente aussagekräftig beschrieben werden können. Darüber hinaus besteht kaum Bedarf an einer Dokumentation, da viele Anliegen in der GraphQL-Spezifikation festgelegt oder durch einen GraphQL-Client übernommen werden.

Auch in gRPC werden Kommentare als Dokumentation der generierten Klassen und Methoden verwendet. Somit ist es aus der IDE heraus möglich, Informationen über die Semantik abzufragen. Die Signatur der Methoden wird in der Schnittstellendefinition festgelegt, sodass keine Dokumentation in Prosaform nötig ist.

	Int. REST	REST	GraphQL	gRPC
Hilfreiche Dokumentation	++	++	+++	+++

[6] Siren: A Hypermedia Specification for Representing Entities (Repository): https://github.com/kevinswiber/siren (besucht am 07.09.2020)

5.3.2. Heuristiken für das Vorbeugen von Fehlern

Eine API sollte so entworfen werden, dass es schwierig ist, sie falsch zu benutzen. Fehleranfällige Situationen sollten eliminiert oder vorher durch die Benutzerin oder den Benutzer explizit bestätigt werden [87]. Die API sollte Benutzerinnen und Benutzer hin zur korrekten Benutzung führen [78]. Bloch formuliert es wie folgt:

> APIs should be easy to use and hard to misuse. It should be easy to do simple things; possible to do complex things; and impossible, or at least difficult, to do wrong things. [93]

Ungültige Eingaben vermeiden

Das Prinzip *Garbage In, Garbage Out* besagt, dass ungültige Eingaben meist zu schlechten Ausgaben führen. Deshalb ist es wichtig, ungültige Eingaben von vornherein möglichst zu vermeiden [94, S. 112].

Für Introspected-REST- und REST-APIs ist ein Schema nicht verpflichtend. Folglich kann es APIs geben, in welchen keine clientseitige Validierung der Eingaben vorgenommen werden kann, da der Client die Erwartungen des Servers nicht kennt. Für beide API-Stile könnte ein *OpenAPI*-Dokument[7] zur Verfügung gestellt werden, in welchem Requests und Responses genau beschrieben werden. Ebenfalls können geeignete Micro- und Mediatypes durch Introspection bzw. durch Hypermedia-Elemente entsprechende Informationen bereitstellen, z.B. JSON Schema [39; 40]. Durch Microtypes wird es aber vereinfacht, ein Schema für die API herauszugeben, da solch ein Microtype nur einmal definiert und dann in Kombination mit anderen Microtypes verwendet werden kann. Monolithische Mediatypes erfordern, dass die Festlegung der Art, wie ein Schema ausgeliefert werden soll, in der Spezifikation des Mediatypes erfolgt.

Sowohl GraphQL als auch gRPC legen genau fest, welche Ein- und Ausgaben erwartet werden. Dadurch können fehlerhafte Eingaben schon im Client erkannt werden. Durch das Typsystem kann vermieden werden, dass illegale Zustände überhaupt erzeugt werden.

	Int. REST	REST	GraphQL	gRPC
Ungültige Eingaben vermeiden	+	–	+++	+++

Beschränkung der zulässigen Aktionen

Durch Beschränkungen werden die möglichen Interaktionen mit einem System begrenzt, wenn diese im aktuellen Zustand nicht zur Verfügung stehen. Es werden potentielle Fehlerquellen reduziert, da Benutzerinnen und Benutzer nicht versuchen, solch unzulässige Interaktionen auszuführen [94, S. 60].

[7] OpenAPI Initiative: https://www.openapis.org/ (besucht am 10.09.2020)

In Introspected REST und REST ist es möglich, Links zu zulässigen Aktionen anhand des aktuellen Status ein- und auszublenden. Verfügt eine Benutzerin oder ein Benutzer bspw. nicht über die nötigen Berechtigungen, um eine Ressource anzufragen, kann der Link zu dieser Ressource einfach nicht gesendet werden.

In Introspected REST ist darauf zu achten, dass die Berechtigungen auch für Introspection-Requests überprüft werden. Probleme bereiten Listen in Introspected REST, wenn einzelne Elemente unterschiedliche Zugriffsberechtigungen erfordern, da der Introspection-Response meist für alle Elemente gilt.

In GraphQL ist es nicht möglich, unzulässige Aktionen zu verbergen. Stattdessen wird z.B. `null` oder eine Fehlermeldung zurückgegeben. In gRPC antwortet der Server mit dem Fehlercode 7 `PERMISSION_DENIED`.

	Int. REST	REST	GraphQL	gRPC
Beschränkung zulässiger Aktionen	++	+++	–	–

5.3.3. Heuristiken für Einfachheit

Einfachheit ist wahrscheinlich die Eigenschaft, die am stärksten mit Benutzerfreundlichkeit verbunden ist und generell als Gegensatz zur Komplexität angesehen wird. Die API sollte über den benötigten Funktionsumfang, trotzdem aber über eine einfache Schnittstelle verfügen.

So klein wie möglich, so groß wie nötig

Die API sollte keine irrelevanten Informationen liefern und generell so klein und einfach gehalten sein wie möglich. Weniger ist mehr [78; 87, S. 115ff.].

Durch die Trennung von Nutz- und Metadaten sowie Content-Negotiation ermöglicht Introspected REST den Clients, relativ genau die gewünschten Informationen auszuwählen. Bei GraphQL bekommen Clients exakt die Daten zurück, welche sie auswählen. REST und gRPC bieten solche Möglichkeiten nicht.

	Int. REST	REST	GraphQL	gRPC
So klein wie möglich	++	–	+++	–

Kleine Arbeitsschritte

Benutzerinnen und Benutzer sollten nur kleine und lokale Änderungen durchführen müssen, um ein Ziel zu erreichen [82]. Andererseits erhöht sich mit der Anzahl der Schritte, die zur Bewältigung einer Aufgabe durchgeführt werden müssen, auch die *kinematische Belastung* [94, S. 178].

Mit Introspected REST und REST können komplexe Workflows in kleine Schritte aufgeteilt werden. Den nächsten Schritt erreicht man durch Auslösen eines Hypermediaelements. Eine fortgeschrittene Technik ist das Partial-Submit-Pattern [102]. Die kinematische Belastung wird in Introspected REST erhöht, da zusätzliche Requests für Introspection nötig sind. Ebenfalls erfolgt ggf. eine reaktive Content-Negotiation, welche nochmals zusätzlichen Aufwand bedeutet.

In GraphQL und gRPC könnten Workflows zwar durch die Hintereinanderausführung mehrerer Mutations bzw. Methoden umgesetzt werden, allerdings ist dieser Prozess fehleranfällig, da der Server den Prozess nicht lenken kann. In GraphQL und gRPC sind komplexe Workflows generell sehr selten vorzufinden.

	Int. REST	REST	GraphQL	gRPC
Kleine Arbeitsschritte	++	++	–	–

Layering

Indem Informationen in Schichten organisiert werden, verbessert sich die Benutzbarkeit. Beispielsweise können nur bestimmte Schichten oder Gruppierungen zu einem Zeitpunkt angezeigt werden, sodass der Fokus auf das Wesentliche gelenkt wird. Durch Schichten können Details verborgen werden [94, S. 146].

Introspected REST und REST ermöglichen Layering durch Links. Ein Übersichts-Zustand kann auf detailreichere Zustände verweisen und gleichzeitig durch die Link-Relation diese Zustände allgemein beschreiben. Beispielsweise kann eine Liste an Notizen nur die wichtigsten Informationen wie ID und Titel darstellen, aber auch einen Link zur Detailansicht einer Notiz bereitstellen. Befindet sich der Client in der Detailansicht, werden gerade nicht benötigte Informationen ausgeblendet.

Layering findet bei GraphQL während der Auswahl der benötigten Daten statt. Objekte dienen als Container für ihre Felder und ermöglichen den Benutzerinnen und Benutzern, abstrakt über diese Dinge nachzudenken. Trotzdem bleiben alle Felder des Selection-Sets sichtbar. In gRPC findet Layering nicht statt.

	Int. REST	REST	GraphQL	gRPC
Layering	+++	+++	+	–

5.3.4. Heuristiken für Konsistenz

Norman argumentiert, dass Komplexität der Welt inhärent und deshalb weder gut noch schlecht ist. Komplexität soll nicht verringert, sondern organisiert werden. Man muss die Verständlichkeit (engl.: *understandability*) verbessern und Unordnung (engl.: *confusion*) vermeiden. Das System muss dafür eine logische Struktur aufweisen und einfach erlernbar sein [103]. Damit das Erlernte auch angewendet werden kann, ist

es wichtig, die API konsistent zu halten. Die Konsistenz wird dabei stark von den Entwicklerinnen und Entwicklern der API beeinflusst, bspw. durch eine konsistente Namensgebung. Aber auch der API-Stil kann darauf Einfluss nehmen.

Erwartungen erfüllen

Wenn die Erwartungen der Benutzerinnen und Benutzer erfüllt und Überraschungen vermieden werden, können diese ihr vorhandenes Wissen anwenden und das Vertrauen in die API steigt [87, S. 132ff.]. Sie können explorativ auf die Interaktionsweise schließen, ohne die Dokumentation bemühen zu müssen, und die API so selbst erkunden.

Introspected REST und REST bieten zwar die Möglichkeit, Hinweise über die Erwartungen durch Hypermedia-Elemente zu kommunizieren, allerdings ist ein Server nicht gezwungen, die Erwartungen zu erfüllen. Überraschungen sind somit möglich.

Durch Content-Negotiation für Microtypes können Clients bei Introspected REST genauer als bei REST spezifizieren, welche Daten sie erhalten möchten und welches Format diese haben sollen. Unterstützt der Server die entsprechenden Microtypes, können Clients somit genauer ihre Erwartungen spezifizieren. In REST hängt es stark vom verwendeten Mediatype ab, wie genau die Erwartungen sein können.

GraphQL zielt darauf ab, den Benutzerinnen und Benutzern genau die Daten zurückzugeben, welche sie anfordern. Damit können jene ihre Erwartungen genau formulieren und diese werden erfüllt, sollte kein Fehler auftreten.

In gRPC werden die Erwartungen durch die Schnittstellendefinition festgelegt, welche der Server erfüllen muss.

	Int. REST	REST	GraphQL	gRPC
Erwartungen erfüllen	++	+	+++	++

Wiedererkennbare Muster

Wiedererkennbare Muster kommunizieren eine bereits bekannte Struktur. Benutzerinnen und Benutzer können so bereits Erlerntes anwenden und die Einarbeitungszeit verringern [104, S. 14f.].

Microtypes ermöglichen ein Feature-Ökosystem für APIs (vgl. Abschnitt 3.1.4). Idealerweise werden Microtypes für verschiedene Funktionalitäten standardisiert und von vielen APIs verwendet. Durch die Standardisierung wird vermieden, dass gleiche Funktionalitäten eine vollkommen verschiedene Schnittstelle aufweisen. Indem Introspected-REST-APIs die gleichen Microtypes verwendet, können sich Benutzerinnen und Benutzer an diese Muster gewöhnen.

In REST gibt es Muster, die allerdings nur als Anhaltspunkte für API-Designerinnen und -Designer dienen. Als Beispiel sei die Auflistung von Bishop[8] genannt. REST und Introspected REST verwenden beide oft das bekannte HTTP-Protokoll und die darauf aufbauende Infrastruktur. Es gibt viele Standards, z.B. für Authentifizierung oder Fehlermeldungen. Verschiedene Mediatypes geben einer API mal mehr, mal weniger Vorgaben, sodass die Wiedererkennbarkeit abhängig von diesem ist. Für alle APIs, welche HAL verwenden, kann bspw. ein generischer Client, der *HAL-Browser*[9], verwendet werden.

Da GraphQL eine detaillierte Spezifikation ist, sind alle APIs sehr einheitlich. Benutzerinnen und Benutzer, die Erfahrung mit GraphQL haben, finden sich in einer anderen GraphQL-API prinzipiell sofort zurecht. Der Bedarf an API-übergreifenden Pattern ist dadurch geringer. Trotzdem existieren einige, bspw. für *Global Object Identification* [105], um Caching durch GraphQL-Clients wie *Relay*[10] oder *Apollo Client*[11] zu ermöglichen [105].

In gRPC existieren wenige Muster bzw. Best Practices. Allerdings zieht sich durch den gesamten Service *ein* wiedererkennbares Muster: Es wird eine Methode mit den benötigten Parametern aufgerufen und es wird eine Antwort zurückgegeben, ggf. als Stream. Der Client-Stub kann in vielen Programmiersprachen generiert werden. Somit können Benutzerinnen und Benutzer ihr bereits vorhandenes Wissen nutzen. Außerdem verwendet der generierte Code bspw. in C# idiomatische Konstrukte der Programmiersprache [106, 06:42–17:56].

	Int. REST	REST	GraphQL	gRPC
Wiedererkennbare Muster	++	+	+++	+

5.3.5. Zusammenfassung

In diesem Abschnitt wurden Heuristiken für die Benutzbarkeit einer API aus Sicht einer Cliententwicklerin oder eines Cliententwicklers aufgestellt. Diese Heuristiken wurden auf die untersuchten API-Stile angewandt und es wurde eine Bewertung durchgeführt. Die Ergebnisse sind in Tabelle 5.7 zusammengefasst.

[8] Matt Bishop. *Patterns – Level 3 REST*: https://level3.rest/patterns (besucht am 07.09.2020)
[9] HAL-Browser (Repository): https://github.com/mikekelly/hal-browser (besucht am 07.09.2020)
[10] Relay – The production-ready GraphQL client for React: https://relay.dev/ (besucht am 07.09.2020)
[11] Apollo Client (React): https://www.apollographql.com/docs/react/ (besucht am 07.09.2020)

	Int. REST	REST	GraphQL	gRPC
Einfacher Einstieg	++	++	+++	+
Sichtbarkeit von Aktionen	++	++	++	+
Aussagekräftige Fehlermeldungen	+	+	++	+
Hilfreiche Dokumentation	++	++	+++	+++
Ungültige Eingaben vermeiden	+	–	+++	+++
Beschränkung zulässiger Aktionen	++	+++	–	–
So klein wie möglich	++	–	+++	–
Kleine Arbeitsschritte	++	++	–	–
Layering	+++	+++	+	–
Erwartungen erfüllen	++	+	+++	++
Wiedererkennbare Muster	++	+	+++	+

Tabelle 5.7.: Zusammenfassung der heuristischen Bewertung der API-Stile

6. Diskussion

In diesem Kapitel werden die Ergebnisse aus Kapitel 5 ausgewertet und diskutiert. Dazu werden vor allem die Auswirkungen von Introspection und Microtypes auf die Performance, Evolvierbarkeit und Komplexität einer Introspected-REST-API beleuchtet.

6.1. Performance

Die Null-Hypothese des Experiments lautete: „Die ermittelte Performance der API ist bei allen vier untersuchten API-Stilen gleich“. Diese Aussage wurde durch die Messergebnisse falsifiziert, da signifikante Unterschiede der Antwortzeiten festgestellt wurden. Generell kann eine starke Korrelation zwischen der Gesamtgröße ausgetauschter Nachrichten und der Antwortzeit festgestellt werden (durchschnittlicher Korrelationsfaktor $\text{corr}(Len_{Req} + Len_{Resp}, RT) = 0,93$. Dies war nach (4.2) und (4.3) zu erwarten, da Latenz und maximale Datenübertragungsrate während des Experiments konstant gehalten wurden.

GraphQL bietet eine exakte Auswahl der Daten und vermeidet so Over- und Underfetching; gRPC setzt mit Protobuf ein effizientes Datenübertragungsformat ein und das ganze Protokoll ist auf eine hohe Performance ausgerichtet [31]. Introspected REST und REST können hingegen nicht mit derartigen Performancefeatures aufwarten. Stattdessen werden mit jedem Response zusätzliche Metadaten gesendet, sodass sich z.B. bei Aufgabe **T1** die Größe des Introspected-REST- und des GraphQL-Response um zwei Größenordnungen unterscheiden. Da die Unterschiede sehr groß ausfallen, stellt sich die Frage, ob GraphQL durch die Auswahl der Aufgaben bevorzugt wurde. Tatsächlich führen Aufgaben **T1** und **T3** zu einem *N+1-Problem* genannten Muster, da für alle n Elemente einer Liste ein oder mehrere zusätzliche Requests gesendet werden müssen, während bei GraphQL alle Informationen in einem Response enthalten sind. Diese Tatsache spricht aber nicht gegen das durchgeführte Experiment, da dies auf die Eigenheiten der API-Stile zurückzuführen sind und ein derartiges Szenario auch in der Realität auftreten kann.

Dass alle Requests innerhalb eines Threads von JMeter sequentiell ausgeführt werden, scheint auf Kosten der API-Stile, die viele Requests benötigen, zu gehen. Beispielsweise könnten bei Aufgabe **T1** alle Notizen durch den Introspected-REST-Client parallel geladen werden. Gerade das Multiplexing in HTTP/2 könnte einen großen Beitrag dazu leisten, die Antwortzeiten von Introspected REST und REST an die von GraphQL anzunähern. Eine solche Untersuchung wurde in der vorliegenden Arbeit aber nicht durchgeführt. Andererseits ist auffällig, dass Introspected REST eine ähnliche Performance wie REST aufweist, obwohl mehr Requests gesendet wurden.

Da ungefähr die gleiche Anzahl an Bytes bei Introspected REST und REST

ausgetauscht werden, weisen auch die Antwortzeiten ähnlich hohe Werte auf. Ein von Vasilakis identifiziertes Problem von REST war die niedrige Performance (Problem **P3**). Für einen hypermediaaffinen Client führt Introspected REST aber zu keiner Verbesserung. Ein anderes Bild würde sich wahrscheinlich ergeben, wenn die Betrachtung für einen Client, der kein Hypermedia verwendet, durchgeführt werden würde. Beispielsweise könnten so bei Aufgabe **T1** 2419 gesendete und 7956 empfangene Bytes eingespart werden. In diesem Experiment wurde nicht untersucht, ob eine exaktere Auswahl der Daten, welche der Server zurücksenden soll, durch Microtypes die Performance beeinflusst. Aspekte sind hierbei, dass eventuell nutzlose Daten nicht zum Client gesendet werden, andererseits aber die Cachebarkeit der Responses verringert wird. Es haben weiterhin Server-Metriken wie der Durchsatz keine Berücksichtigung gefunden. Auch wurde die vom Client wahrgenommene Performance nicht berücksichtigt: Werden nämlich viele kleine Requests gesendet, kann der Client schon die Daten aus den ersten erhaltenen Responses anzeigen, während er noch auf andere wartet. So scheint der Client die Arbeit schon verrichtet zu haben, obwohl diese noch nicht vollständig abgeschlossen ist.

6.2. Evolvierbarkeit

REST ist ein API-Stil, welcher auf Langlebigkeit ausgelegt ist [74, S. 41], und Introspected REST erbt diese Eigenschaft. Von den beiden Grundpfeilern von Introspected REST, dem Introspection-Prinzip und Microtypes, wurde angenommen, dass vor allem Microtypes die Chance eröffnen, die Evolvierbarkeit einer API positiv zu beeinflussen, denn das Introspection-Prinzip verändert nur den Ort der Hypermedia-Elemente. Introspection verbessert die Evolvierbarkeit dahingehend, dass Hypermedia zu einer bestehenden API hinzugefügt werden kann, ohne den für die Nutzdaten verwendeten Mediatype oder bestehende Clients anpassen zu müssen. Bei der NeverNote-API wurde Hypermedia allerdings von Beginn an verwendet.

Bei dem Vergleich in Abschnitt 5.2 unterscheiden sich Introspected REST und REST nur bei Änderung **C9**. Tatsächlich dienen Microtypes auch eher dazu, Details der Kommunikation wie das Datenaustauschformat zu verhandeln oder bestimmte Funktionalität anzufordern, und beschreiben die Semantik ihrer eigenen, kleinen Schnittstelle [42]. Microtypes erhöhen die Austauschbarkeit der Funktionalität auf Clientseite, da sie abgeschlossene, zusammensetzbare Teile eines Ganzen (Holons) sind. Zum Beispiel können *HAL-FORMS*[1] durch *JSONForms*[2] ohne Auswirkungen auf andere Microtypes ersetzt werden. Doch bietet ein Server einen Microtype einmal an, gehört dieser zur öffentlichen Schnittstelle der API und unterliegt den gleichen Einflüssen hinsichtlich der Evolvierbarkeit wie jeder andere Teil der Schnittstelle.

[1] The HAL-FORMS Media Type: https://rwcbook.github.io/hal-forms (besucht am 14.09.2020)

[2] JSONForms: https://jsonforms.io/ (besucht am 14.09.2020)

Tatsächlich wird durch die Unterstützung vieler Microtypes die API-Oberfläche größer, sodass insgesamt sogar mehr abwärtsinkompatible Änderungen auftreten könnten. Abwärtsinkompatible Änderungen würden eine neue Version eines Microtypes nach sich ziehen, sodass bestehende Clients nicht in den Genuss der Änderungen kommen. Microtypes verbessern also die Evolvierbarkeit der NeverNote-API nur in geringem Maße. Sie erhöhen die Austauschbarkeit für Clients, aber nicht die Änderbarkeit.

Introspected REST und REST erreichen Evolvierbarkeit vor allem durch eine lose Kopplung von Client und Server. Genauer gesagt wird Businesslogik nicht im Client dupliziert. So können Änderungen erfolgen, ohne dass viele Clients angepasst werden müssen. Stattdessen adoptieren sie die Änderungen automatisch, sobald der Server die geänderten Informationen bereitstellt [107, 37:35–38:47]. Dadurch wird erreicht, dass bspw. bei Änderung **C7** keine Anpassung der Introspected-REST- und REST-Clients erforderlich ist, während dies bei GraphQL und gRPC der Fall ist. Im Vergleich zu gRPC stellen Introspected REST, REST und GraphQL eine Möglichkeit bereit, Informationen über die API durch Hypermedia und/oder Introspection zur Laufzeit abzufragen. Clients können automatisch angepasst werden, indem sie die Laufzeitinformationen verarbeiten, anstatt auf zur Übersetzungszeit kodierte Informationen zu vertrauen. gRPC-Clients verfügen bspw. nur über das Wissen aus einer vorher festgesetzten Servicedefinition und können dieses nicht aktualisieren. Eine automatische Anpassung des Clients, bspw. bei Änderung **C1**, ist somit nicht möglich.

6.3. Komplexität

Wie in Abschnitt 4.3.2 beschrieben, wurde bewusst auf eine Gewichtung der Heuristiken verzichtet, sodass von einer abschließenden Bewertung durch Addition der einzelnen Ergebnisse abzusehen ist. Weiterhin sei noch einmal betont, dass es sich nur um Heuristiken handelt und die Auflistung unvollständig ist. Andere Heuristiken könnten bspw. bestimmen, wie schnell die API einer Cliententwicklerin oder einem Cliententwickler Feedback geben kann oder ob Abkürzungen für Power-Userinnen und -User eingeführt werden können, um die Effizienz oft verwendeter Aktionen zu steigern,

Introspection verbessert die Benutzbarkeit der Introspected-REST-API vor allem für Clients, welche nicht an Hypermedia interessiert sind. Diesen bleiben unnötige Informationen erspart. Auch Microtypes führen dazu, dass exakter ausgewählt werden kann, welche Informationen ein Client vom Server bekommen will. So kann er sich z.B. zwischen verschiedenen Introspection-Microtypes entscheiden, während solch fortgeschrittene Content-Negotiation bei REST-APIs nicht üblich ist. Würde die Utopie eines Microtype-Ökosystems (siehe Abschnitt 3.1.4) Realität werden, würden Microtypes auch erheblich zur Konsistenz zwischen APIs verschiedener Anbieter

beitragen. Es wäre wohl der Traum einiger App-Entwicklerinnen und -Entwickler, wenn mehrere soziale Netzwerke über eine strukturell gleiche API verfügen würden und man den gleichen Client wiederverwenden könnte, bestenfalls sogar nur die Einstiegs-URL anpassen müsste. Ein weiterer positiver Aspekt der Microtypes ist die Modularität, wodurch theoretisch Softwaremodule für einzelne Microtypes für Server und Client bereitgestellt und durch die Wiederverwendung die Programmierung erleichtert werden könnte.

Einen Einfluss auf die Benutzbarkeit einer Introspected-REST-API konnte der Autor auch durch die Trennung von Nutzdaten und Hypermedia feststellen, da die beiden Responses für die Verwendung einiger Hypermediaelemente wie etwa URI-Templates ([108]) zusammengesetzt werden mussten. Gefühlt war durch die Aufspaltung in zwei separate Responses mehr Arbeit nötig. Ein weiterer Punkt ist die gefühlte Komplexitätssteigerung durch die fortgeschrittene Content-Negotiation bei Introspected REST. Anstatt eines einzelnen Mediatypes, welcher während der Nutzung der API meist konstant bleibt, müssen nun im `Accept`-Header viele Parameter zur Auswahl der Microtypes angegeben werden.

Alle untersuchten API-Stile haben ihre Eigenheiten, wobei manche der API zum Vorteil, andere zum Nachteil gereichen. Für individuelle Entwicklerinnen und Entwickler ist das Hauptargument wahrscheinlich die Vertrautheit mit dem jeweiligen API-Stil. Würde eine Programmiererin oder ein Programmierer heute mit einer Introspected-REST-API konfrontiert werden, könnte sie oder er die API wahrscheinlich grundlegend verwenden, da sie auf HTTP und REST aufbaut. Doch bis sie sich an Introspection gewöhnt haben, bedürfte es wohl einer Eingewöhnungsphase.

6.4. Zusammenfassung

Die beiden Grundpfeiler von Introspected REST, das Introspection-Prinzip und Microtypes, wirken sich vor allem auf die Performance und Komplexität/Benutzbarkeit einer API aus. Durch die Trennung von Nutz- und Metadaten werden bei einem hypermediaaffinen Client für jeden REST-Request zwei Introspected-REST-Requests benötigt. Dies erhöht die Anzahl gesendeter Bytes, doch verhalten sich die Performanceergebnisse weiterhin ähnlich zu denen der REST-API. Die Auswirkungen von Introspection auf die Benutzbarkeit sind zwiespältig. Einerseits ermöglichen sie es einem Client, sich auf die Nutzdaten bzw. die Metadaten zu fokussieren und letztere ggf. zu ignorieren. Andererseits erhöht die Trennung die kognitive Belastung. Microtypes ermöglichen einem Client, genauer auszuwählen, welche Daten der Server zurückschicken soll, und Erwartungen an diese Daten zu formulieren. Sie eröffnen ebenfalls die Möglichkeit, die Konsistenz innerhalb einer API und zwischen APIs sicherzustellen. Die Evolvierbarkeit wird durch Introspection für eine API erhöht, die zuvor ohne Hypermedia auskommen musste. Microtypes erhöhen nur die Austauschbarkeit von Funktionalität der API, aber nicht deren Änderbarkeit.

7. Fazit und Ausblick

Diese Arbeit hat sich mit Introspected REST, einem neuen Architekturstil, welcher auf REST aufbaut, beschäftigt. Introspected REST übernimmt die sechs REST-Prinzipien, konkretisiert das Prinzip „Hypermedia as the Engine of Application State“ aber durch das *Introspection-Prinzip* „Introspection as the Engine of Application State“. Dieses besagt, dass Nutzdaten und statische Metadaten, allen voran Hypermedia, getrennt werden sollen. Clients stellen also einen Request, um die Nutzdaten abzufragen, und einen zweiten, um an die Metadaten zu gelangen. Neben dem Introspection-Prinzip bilden *Microtypes* einen weiteren Grundpfeiler von Introspected REST. Während die Mediatypes von REST-APIs meist monolithisch aufgebaut sind und die komplette Funktionalität der API beschreiben, soll der verwendete Mediatype einer Introspected-REST-API aus unabhängigen Bausteinen, den Microtypes, zusammengesetzt werden. Ein Microtype beschreibt nur einen kleinen Teil der API-Funktionalität, z.B., wie eine Liste von Elementen gefiltert wird oder wie Fehlermeldungen ausgegeben werden. Die verwendeten Microtypes werden zwischen Client und Server durch Content-Negotiation ausgehandelt. Dies ermöglicht einem Client, genauer als bei REST-APIs zu bestimmen, welche Daten der Server zurückgeben soll, und die gleiche API kann von einer Vielzahl von Clients verwendet werden. Eine weitere Erleichterung für Clients könnte durch die API-übergreifende Verwendung standardisierter Microtypes erreicht werden, denn so erhöht sich die Konsistenz zwischen verschiedenen APIs und die Wiederverwendung von Clientcode wird ermöglicht.

Introspected-REST wurde von Filippos Vasilakis, dem Erfinder von Introspected REST, als Alternative zu heutzutage verbreiteten API-Stilen positioniert. Um diese Behauptung zu untersuchen, wurde ein Vergleich von Introspected REST mit REST sowie GraphQL, einer Datenabfragesprache, und gRPC, einem Protokoll für effiziente Interprozesskommunikation, gezogen. Diese drei alternativen API-Stile werden heutzutage am meisten verwendet. Als Vergleichskriterien wurden die Performance, die Evolvierbarkeit sowie die Komplexität/Benutzbarkeit gewählt.

Die Performance wurde aus der Perspektive eines Clients bewertet. Dazu wurden APIs zur Verwaltung von Notizen mit den vier zu vergleichenden API-Stilen erstellt. Ein oder mehrere Clients mussten nun Anwendungsfälle durchspielen. Für jeden Anwendungsfall wurde die benötigte Zeit sowie die Anzahl ausgetauschter Bytes gemessen. Die Messergebnisse zeigten, dass die Introspected-REST- und REST-APIs sehr „geschwätzig“ waren und viele Daten zwischen Client und Server ausgetauscht wurden, während GraphQL und gRPC mit einer kleineren Datenmenge zurechtkamen. Entsprechend benötigten die Introspected-REST- und REST-Clients auch mehr Zeit, um einen Anwendungsfall abzuschließen. Eine Verbesserung von Introspected REST gegenüber REST hinsichtlich der Performance konnte nicht festgestellt werden – zumindest nicht bei einem Client, welcher Hypermedia anfordert.

Für den Vergleich der Evolvierbarkeit wurden veränderte Anforderungen an eine API simuliert. Für jeden API-Stil wurde beschrieben, ob eine Änderung abwärtskompatibel ist, d.h., ob der Client ohne Anpassungen weiterhin mit dem Server kommunizieren kann, und, falls nicht, warum Anpassungen erforderlich sind. Es zeigte sich, dass Introspected REST und REST mit Änderungen besser zurechtkommen als GraphQL und gRPC, da Businesslogik vom Server kontrolliert und nicht im Client dupliziert wird. Es konnten nur sehr geringe Unterschiede zwischen Introspected REST und REST bei dieser Untersuchung festgestellt werden. Durch Microtypes können Clients zwar einen Microtype leicht durch einen anderen ersetzen, doch beeinflusst dies nicht, wie gut ein individueller Microtype weiterentwickelt werden kann. Die Evolvierbarkeit wird durch Introspection dahingehend erhöht, dass APIs, welche zuvor Hypermedia nicht verwendet haben, dies nun tun können, ohne Änderungen an bestehenden Clients zu fordern.

Die Komplexität wurde anhand von Heuristiken für die Benutzbarkeit einer API bewertet. Für jeden API-Stil wurde festgestellt, wie gut dieser eine Heuristik realisiert. Es ergab sich dabei ein gemischtes Bild: Jeder API-Stil zeigte individuelle Stärken und Schwächen. Introspected REST blieb vor allem im Bereich der Fehlervermeidung zurück. GraphQL und gRPC ermöglichen z.B. durch einen expliziten Schnittstellenvertrag eine bessere Validierung der Eingaben von Benutzerinnen und Benutzern im Client. Dafür verbessern vor allem Microtypes die Konsistenz einer Introspected-REST-API im Vergleich zu REST.

Zusammenfassend lässt sich sagen, dass Introspected REST versucht, einen pragmatischen Mittelweg zu gehen zwischen einer geringeren Komplexität und trotzdem guter Evolvierbarkeit. Pragmatisch deshalb, weil in der Realität viele Clients von REST-APIs Hypermedia nicht verwenden und stattdessen URLs und Businesslogik hart kodieren. Eine Introspected-REST-API verbessert die Performance und Benutzbarkeit für solche Clients, da sie nicht mehr genötigt werden, mit Hypermedia umzugehen. Ebenfalls gibt Introspected REST durch die Idee der Microtypes den API-Entwicklerinnen und -Entwicklern ein Werkzeug an die Hand, um mit der Herausforderung vieler verschiedener Anforderungen durch eine Vielzahl von Clients umzugehen. Der einzige API-Stil, der sonst einen konkreten Ansatz in dieser Hinsicht bietet, ist GraphQL. Doch existieren drei Jahre nach der Präsentation von Introspected REST immer noch keine öffentlichen Microtypes. Um eine Adoption des API-Stils zu realisieren, wird aber eine kritische Masse dieser benötigt, sodass ein API-Anbieter nicht die Last des Designs aller benötigten Microtypes allein tragen muss. Bisher wird auch noch kein Prozess für die Standardisierung von Microtypes vorgeschlagen, sodass hier die Gefahr besteht, dass jeder sein eigenes Süppchen kocht – sollte Introspected REST eine messbare Verbreitung erreichen. Das Microtype-Ökosystem ist jedenfalls noch in weiter Ferne.

Zwar dauerte es auch einige Jahre, bis REST als API-Stil Verwendung fand. Doch klang REST zu diesem Zeitpunkt wie eine willkommene Alternative zu SOAP,

welches zu einer engen Kopplung zwischen Client und Server führte, XML verwendete und nicht gemacht war für das Web. Eine derartige Situation liegt heute nicht vor. API-Anbieter haben die Wahl zwischen REST für langlebige APIs, GraphQL für datenintensive, clientzentrierte APIs, gRPC für interne APIs und anderen HTTP-basierten APIs, die oft fälschlicherweise als „REST APIs“ bezeichnet werden, falls sie den für sie bequemen und bekannten Ansatz wählen wollen. Und dann gibt es noch viele andere, weniger verbreitete API-Stile. Introspected REST muss sich also gegen eine Vielzahl von Konkurrenten durchsetzen. Wenn man als API-Anbieter nun verschiedene Gruppen von Entwicklerinnen und Entwicklern bedienen muss, vor allem solche, die einen Client, der langfristig wenige Anpassungen erfordert, bauen wollen, und solche, die einen HTTP-Client in der Weise erstellen möchten, wie sie es gewohnt sind, kann Introspected REST theoretisch ein guter Ansatz sein. Doch da bis jetzt keine Infrastruktur dafür vorhanden ist, gleichen die Vorteile von Introspected REST den initialen Aufwand wahrscheinlich nicht aus.

In dieser Arbeit wurden Introspected-REST-Clients als hypermediaaffin angenommen. Es bleibt offen, wie vorteilhaft der API-Stil für Clients ohne Hypermediaverwendung ist. Eine weitere interessante Frage ist, ob Introspected-REST- und REST-APIs mit feingliedriger Aufteilung der Ressourcen durch den Einsatz von HTTP/2 eine ähnlich gute Performance erreichen können wie GraphQL. Die Beschreibung von Introspected REST durch Vasilakis empfand der Autor als nicht sehr detailreich und offen für Interpretationen, die sich in der Implementierung widerspiegeln. Es ist wünschenswert, dass diese Interpretationsspielräume geschlossen oder konkurrierende Interpretationen empirisch untersucht werden. Eine Lehre aus Introspected REST ist, dass Mediatypes in der jetzigen Form nicht ideal sind und darüber nachgedacht werden sollte, wie man sie änderbarer gestaltet. Microtypes sind ein möglicher Weg, doch könnten während der Erforschung des Problems sicherlich alternative oder sogar bessere Ansätze gefunden werden.

Es bleibt also weiterhin Bewegung im Web-API-Bereich und auch Introspected REST bildet nicht das Ende der Geschichte.

Literatur

[1] L. Richardson, „Justice Will Take Us Millions Of Intricate Moves,“ in *QCon San Francisco 2008*, [Konferenzbeitrag], 20. Nov. 2008.

[2] M. Fowler. (14. Dez. 2006). „SemanticDiffusion.“ [Online], URL: https://www.martinfowler.com/bliki/SemanticDiffusion.html (besucht am 14. 09. 2020).

[3] „Gartner Hype Cycle,“ Gartner Inc., [Online]. URL: https://www.gartner.com/en/research/methodologies/gartner-hype-cycle (besucht am 14. 09. 2020).

[4] A. Lauret, *The Design of Web APIs*. Shelter Island, NY: Manning, 2019.

[5] B. Jin, S. Sahni und A. Shevat, *Designing Web APIs*. Sebastopol, CA: O'Reilly, 2018.

[6] Wikipedia Contributors, *Web API – Wikipedia, The Free Encyclopedia*, [Online], 2020. URL: https://en.wikipedia.org/w/index.php?title=Web_API&oldid=974803869 (besucht am 11. 09. 2020).

[7] D. Jacobson, G. Brail und D. Woods, *APIs: A Strategy Guide*. Sebastopol, CA: O'Reilly, 2011.

[8] E. Porcello und A. Banks, *Learning GraphQL*. Sebastopol, CA: O'Reilly, 2018.

[9] K. Indrasiri und D. Kuruppu, *gRPC: Up and Running*. Sebastopol, CA: O'Reilly, 2020.

[10] R. Mitra, „Developers are People Too! Building a DX based API Strategy,“ in *Nordic APIs*, [Konferenzbeitrag], Stockholm, 18. Sep. 2013.

[11] R. T. Fielding, R. N. Taylor, J. R. Erenkrantz, M. M. Gorlick, J. Whitehead, R. Khare und P. Oreizy, „Reflections on the REST architectural style and "principled design of the modern web architecture",“ in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, ACM Press, 2017.

[12] J. Webber, S. Parastatidis und I. Robinson, *REST in Practice*. Sebastopol, CA: O'Reilly, 2010.

[13] T. Berners-Lee, „Information Management: A Proposal,“ 1989.

[14] T. Berners-Lee, T. Bray, D. Connolly, P. Cotton, R. Fielding, M. Jeckle, C. Lilley, N. Mendelsohn, D. Orchard, N. Walsh und S. Williams, „Architecture of the World Wide Web, Volume One,“ W3C, 15. Dez. 2004.

[15] T. Berners-Lee, R. T. Fielding und L. M. Masinter, „Uniform Resource Identifier (URI): Generic Syntax,“ IETF Secretariat, RFC 3986, Jan. 2005.

[16] R. T. Fielding und R. N. Taylor, „Architectural Styles and the Design of Network-Based Software Architectures,“ Diss., University of California, Irvine, Irvine, 2000.

[17] R. T. Fielding und J. Reschke, „Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,“ IETF Secretariat, RFC 7231, Juni 2014.

[18] N. Freed und N. S. Borenstein, „Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types,“ IETF Secretariat, RFC 2046, Nov. 1996.

[19] L. Richardson und M. Amundsen, *RESTful Web APIs*. Sebastopol, CA: O'Reilly, 2013.

[20] L. M. Masinter und D. W. Connolly, „The 'text/html' Media Type,“ IETF Secretariat, RFC 2854, Jan. 2000.

[21] T. Bray, „The JavaScript Object Notation (JSON) Data Interchange Format,“ IETF Secretariat, RFC 8259, Dez. 2017.

[22] M. Nottingham, „Web Linking,“ IETF Secretariat, RFC 8288, Okt. 2017.

[23] R. T. Fielding. (20. Okt. 2008). „REST APIs must be hypertext-driven.“ [Online], URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 02. 07. 2020).

[24] J. J. Gibson, *The Ecological Approach to Visual Perception*. Boston: Houghton Mifflin, 1979.

[25] L. Bass, P. Clements und R. Kazman, *Software Architecture in Practice*, 3. Aufl., Ser. SEI Series in Software Engineering. Upper Saddle River, NJ: Addison Wesley, 2012.

[26] S. Tilkov, M. Eigenbrodt, S. Schreier und O. Wolf, *REST und HTTP*, 3. Aufl. Heidelberg: dpunkt.Verlag, 2015.

[27] M. Nottingham. (13. Okt. 2019). „How Multiplexing Changes Your HTTP APIs.“ [Online], URL: https://www.mnot.net/blog/2019/10/13/h2_api_multiplexing (besucht am 27. 07. 2020).

[28] L. Byron und Mitwirkende, *GraphQL*, June 2018, [Online], GraphQL Foundation, 10. Juni 2018. URL: http://spec.graphql.org/June2018/ (besucht am 22. 07. 2020).

[29] L. Byron. (14. Sep. 2015). „GraphQL: A data query language.“ [Online], URL: https://engineering.fb.com/core-data/graphql-a-data-query-language/ (besucht am 08. 07. 2020).

[30] M.-A. Giroux, *Production Ready GraphQL*. März 2020.

[31] Google LLC, *About gRPC – gRPC*, [Online]. URL: https://grpc.io/about/ (besucht am 29. 07. 2020).

[32] J. E. White, „High-level framework for network-based resource sharing,“ IETF Secretariat, RFC 707, 14. Jan. 1976.

[33] R. Thurlow, „RPC: Remote Procedure Call Protocol Specification Version 2,“ IETF Secretariat, RFC 5531, Mai 2009.

[34] A. D. Birrell und B. J. Nelson, „Implementing Remote Procedure Calls,“ *ACM Transactions on Computer Systems (TOCS)*, Jg. 2, Nr. 1, S. 39–59, Feb. 1984.

[35] J. Waldo, G. Wyant, A. Wollrath und S. Kendall, „A Note on Distributed Computing,“ Sun Microsystems Laboratories, Inc., Techn. Ber., Nov. 1994.

[36] Google LCC, *gRPC Motivation and Design Principles – gRPC*, [Online]. URL: https://grpc.io/blog/principles/ (besucht am 29. 07. 2020).

[37] Google LLC, *Supported languages and platforms – gRPC*, [Online]. URL: https://grpc.io/docs/languages/ (besucht am 28. 08. 2020).

[38] F. Vasilakis. (10. Sep. 2017). „Introspected REST: An alternative to REST and GraphQL.“ [Online], URL: https://introspected.rest/.

[39] A. Wright, H. Andrews, B. Hutton und G. Dennis, „JSON Schema: A Media Type for Describing JSON Documents,“ IETF Secretariat, Internet-Draft draft-handrews-json-schema-02, 17. Sep. 2019.

[40] A. Wright, H. Andrews und B. Hutton, „JSON Schema Validation: A Vocabulary for Structural Validation of JSON,“ IETF Secretariat, Internet-Draft draft-handrews-json-schema-validation-02, 17. Sep. 2019.

[41] M. Kelly, „JSON Hypertext Application Language,“ IETF Secretariat, Internet-Draft draft-kelly-json-hal-08, 11. Mai 2016.

[42] F. Vasilakis. (23. Nov. 2017). „MicroTypes: Let's break down the monoliths of content negotiation.“ [Online], URL: http://microtypes.info/ (besucht am 22. 06. 2020).

[43] Apache Software Foundation, *Content Negotiation*, [Online]. URL: https://httpd.apache.org/docs/2.4/content-negotiation.html (besucht am 15. 09. 2020).

[44] M. Nottingham und E. Wilde, „Problem Details for HTTP APIs,“ IETF Secretariat, RFC 7807, März 2016.

[45] G. Kellogg, P.-A. Champin und D. Longley, „JSON-LD 1.1,“ W3C, W3C Recommendation, 16. Juli 2020.

[46] H. Andrews und A. Wright, „JSON Hyper-Schema: A Vocabulary for Hypermedia Annotation of JSON,“ IETF Secretariat, Internet-Draft draft-handrews-json-schema-hyperschema-02, 17. Sep. 2019.

[47] L. Plotnicki. (5. Dez. 2015). „BFF @ SoundCloud.“ [Online], ThoughtWorks, Inc., URL: https://www.thoughtworks.com/insights/blog/bff-soundcloud (besucht am 12. 09. 2020).

[48] M.-A. Giroux. (1. März 2019). „Where we Come From: An Honest Introduction to GraphQL.“ [Online], URL: https://medium.com/@__xuorig__/where-we-come-from-an-honest-introduction-to-graphql-4a2ef6124488 (besucht am 12. 09. 2020).

[49] R. Verborgh und M. Dumontier, „A Web API Ecosystem through Feature-Based Reuse,“ *IEEE Internet Computing*, Jg. 22, Nr. 3, S. 29–37, Mai 2018.

[50] E. Wilde, „Surfing the API Web,“ in *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, ACM Press, 2018.

[51] M. Nottingham, „Home Documents for HTTP APIs,“ IETF Secretariat, Internet-Draft draft-nottingham-json-home-06, 15. Feb. 2017.

[52] Microsoft Corp., *C# Language Specification*, 5. Aufl., Standard ECMA-334, Ecma International, Dez. 2017.

[53] L. G. Williams und C. U. Smith, „Performance evaluation of software architectures,“ in *Proceedings of the first international workshop on Software and performance - WOSP '98*, ACM Press, 1998.

[54] C. Loosley und F. Douglas, *High-Performance Client/Server*. New York: Wiley, 1998.

[55] A. Bondi, *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. Upper Saddle River, NJ: Addison-Wesley, 2015.

[56] L. Cherkasova, Y. Fu, W. Tang und A. Vahdat, „Measuring and Characterizing End-to-End Internet Service Performance,“ *ACM Transactions on Internet Technology (TOIT)*, Jg. 3, Nr. 4, S. 347–391, Nov. 2003.

[57] G. Coulouris, J. Dollimore, T. Kindberg und G. Blair, *Distributed Systems: Concepts and Design*, 5. Aufl. Boston: Addison-Wesley, 2012.

[58] C. Vazac und I. Grigorik, „Server Timing,“ W3C, W3C Working Draft, 28. Juli 2020.

[59] A. Håkansson, „Portal of Research Methods and Methodologies for Research Projects and Degree Projects,“ in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering FECS'13*, H. R. Arabnia, A. Bahrami, V. A. Clincy, L. Deligiannidis und G. Jandieri, Hrsg., Las Vegas USA: CSREA Press, 2013, S. 67–73.

[60] R. Voss, *Wissenschaftliches Arbeiten*, 6. Aufl. München: UVK Verlag, 2019.

[61] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell und A. Wesslén, *Experimentation in Software Engineering*. Heidelberg: Springer-Verlag, 2012.

[62] Apache Software Foundation, *Apache JMeter - User's Manual: Glossary*, [Online]. URL: https://jmeter.apache.org/usermanual/glossary.html (besucht am 02. 08. 2020).

[63] ——, *Apache JMeter - User's Manual: Elements of a Test Plan*, [Online]. URL: https://jmeter.apache.org/usermanual/test_plan.html (besucht am 02. 08. 2020).

[64] Y. Katz, D. Gebhardt, G. Sullice und Mitwirkende, *JSON:API*, Version 1.0, [Online], Mai 2015. URL: https://jsonapi.org/format/ (besucht am 29. 08. 2020).

[65] M. I. Landeiro und I. Azevedo, „Analyzing GraphQL Performance," in *Software Engineering for Agile Application Development*, IGI Global, 2020, S. 109–140.

[66] M. Seabra, M. F. Nazário und G. Pinto, „REST or GraphQL?" In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse - SBCARS '19*, ACM Press, 2019.

[67] M. Cederlund, „Performance of frameworks for declarative data fetching, An evaluation of Falcor and Relay+GraphQL," Magisterarb., KTH, School of Information und Communication Technology (ICT), Juli 2016.

[68] T. Eizinger, „API Design in Distributed Systems: A Comparison between GraphQL and REST," Magisterarb., Fachhochschule Technikum Wien, 4. Mai 2017.

[69] K. Gustavsson und E. Stenlund, „Efficient data communication between a webclient and a cloud environment," Magisterarb., Department of Electrical und Information Technology, Faculty of Engineering, Lund University, 23. Juni 2016.

[70] H. P. Breivold, I. Crnkovic und P. Eriksson, „Evaluating Software Evolvability," in *Proceedings of the 7th Conference on Software Engineering Research and Practice in Sweden*, T. Arts, Hrsg., Göteborg, Sweden, Feb. 2007, S. 96–103.

[71] D. Rowe, J. Leaney und D. Lowe, „Defining systems evolvability - a taxonomy of change," in *International Conference and Workshop:Engineering of Computer-Based Systems (ECBS)*, 1998.

[72] L. Xavier, A. Brito, A. Hora und M. T. Valente, „Historical and Impact Analysis of API Breaking Changes: A Large-Scale Study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, Feb. 2017.

[73] R. Michela, „Stop Breaking the Proto! Designing for Change in an Microservices World," in *gRPC Conf 2020*, [Konferenzbeitrag], 28. Juli 2020.

[74] G. Block, P. Cibraro, P. Felix, H. Dierking und D. Miller, *Designing Evolvable Web APIs with ASP.NET*. Sebastopol, CA: O'Reilly, März 2014.

[75] M. Amundsen, *RESTful Web Clients*. Sebastopol, CA: O'Reilly, 2017.

[76] W.-J. Tsaur und S.-J. Horng, „A new generalized software complexity metric for distributed programs," *Information and Software Technology*, Jg. 40, Nr. 5-6, S. 259–269, Juli 1998.

[77] R. Mitra, „A Simpler Time: Balancing Simplicity and Complexity," in *Nordic APIs*, [Konferenzbeitrag], Kopenhagen, 11. Mai 2015.

[78] B. A. Myers und J. Stylos, „Improving API Usability," *Communications of the ACM*, Jg. 59, Nr. 6, S. 62–69, Mai 2016.

[79] D. Saffer, *Designing for Interaction: Creating Innovative Applications and Devices*, 2. Aufl. Berkeley, CA: New Riders, 2010.

[80] J. Nielsen und R. Molich, „Heuristic Evaluation of User Interfaces," in *Proceedings of the SIGCHI conference on Human factors in computing systems Empowering people - CHI '90*, ACM Press, 1990.

[81] T. R. Green und M. Petre, „Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing*, Jg. 7, Nr. 2, S. 131–174, Juni 1996.

[82] S. Clarke und C. Becker, „Using the Cognitive Dimensions Framework to evaluate the usability of a class library," in *In Proceedings of the 15h Workshop of the Psychology of Programming Interest Group (PPIG 2003)*, 2003.

[83] P. G. Polson, C. Lewis, J. Rieman und C. Wharton, „Cognitive walkthroughs: a method for theory-based evaluation of user interfaces," *International Journal of Man-Machine Studies*, Jg. 36, Nr. 5, S. 741–773, Mai 1992.

[84] M. H. Blackmon, P. G. Polson, M. Kitajima und C. Lewis, „Cognitive Walkthrough for the Web," in *Proceedings of the SIGCHI conference on Human factors in computing systems Changing our world, changing ourselves - CHI '02*, ACM Press, 2002.

[85] J. Brooke, „SUS: A 'Quick and Dirty' Usability Scale," in *Usability Evaluation In Industry*, P. W. Jordan, B. Thomas, B. A. Weerdmeester und I. L. McClelland, Hrsg. London: Taylor & Francis, 1996, Kap. 21.

[86] C. Lewis, „Using the Thinking Aloud Method in Cognitive Interface Design," IBM, Techn. Ber., 1982.

[87] J. Nielsen, *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1994.

[88] E. de Kock, J. van Biljon und M. Pretorius, „Usability evaluation methods: Mind the gaps," in *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on - SAICSIT '09*, ACM Press, 2009.

[89] E. L.-C. Law und E. T. Hvannberg, „Analysis of Strategies for Improving and Estimating the Effectiveness of Heuristic Evaluation," in *Proceedings of the third Nordic conference on Human-computer interaction - NordiCHI '04*, ACM Press, 2004.

[90] C. R. B. de Souza und D. L. M. Bentolila, „Automatic evaluation of API usability using complexity metrics and visualizations," in *2009 31st International Conference on Software Engineering - Companion Volume*, IEEE, Mai 2009.

[91] M. F. Zibran, „What Makes APIs Difficult to Use," *International Journal of Computer Science and Network Security*, Jg. 8, Nr. 4, Apr. 2008.

[92] T. Scheller und E. Kühn, „Automated Measurement of API usability: The API Concepts Framework," *Information and Software Technology*, Jg. 61, S. 145–162, Mai 2015.

[93] J. Bloch, „How to design a good API and why it matters," in *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, ACM Press, 2006.

[94] W. Lidwell, K. Holden und J. Butler, *Universal Principles of Design*. Beverly, MA: Rockport Publishers, 2010.

[95] J. Nielsen, *Usability inspection methods*. New York: Wiley, 1994.

[96] E. Mosqueira-Rey, D. Alonso-Ríos, V. Moret-Bonillo, I. Fernández-Varela und D. Álvarez-Estévez, „A Systematic Approach to API Usability: Taxonomy-derived Criteria and a Case Study," *Information and Software Technology*, Jg. 97, S. 46–63, Mai 2018.

[97] M. P. Robillard, „What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software*, Jg. 26, Nr. 6, S. 27–34, Nov. 2009.

[98] M. Piccioni, C. A. Furia und B. Meyer, „An Empirical Study of API Usability," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, IEEE, Okt. 2013.

[99] C. Lilienthal, „Komplexität von Softwarearchitekturen," Diss., Universität Hamburg, Fachbereich Informatik, Juli 2008.

[100] M. Birbeck und S. McCarron, „CURIE Syntax 1.0," W3C, W3C Note, 16. Dez. 2010.

[101] E. Wilde, „The 'profile' Link Relation Type," IETF Secretariat, RFC 6906, März 2013.

[102] M. Amundsen, „12 Patterns for Hypermedia Service Architecture," in *O'Reilly Software Architecture Conference in New York*, [Konferenzbeitrag], New York: O'Reilly, Apr. 2016. URL: https://www.oreilly.com/content/12-patterns-for-hypermedia-service-architecture/.

[103] D. Norman, *Living with complexity*. Cambridge, MA: MIT Press, 2011.

[104] J. Tidwell, *Designing Interfaces: Patterns for Effective Interaction Design*. Sebastopol, CA: O'Reilly, 2011.

[105] Facebook, Inc, *Global Object Identification | GraphQL*, [Online]. URL: https://graphql.org/learn/global-object-identification/ (besucht am 15. 09. 2020).

[106] S. Shirhatti und J. Luo, „Lessons Learned in Building a gRPC Implementation for .NET Core," in *gRPC Conf 2020*, [Konferenzbeitrag], 27. Juli 2020.

[107] O. Drotbohm, „REST beyond the Obvious," in *GOTO Amsterdam 2019*, [Konferenzbeitrag], Amsterdam: GOTO Conferences, 19. Juni 2019. URL: https://gotoams.nl/2019/sessions/802/rest-beyond-the-obvious-api-design-for-ever-evolving-systems.

[108] R. T. Fielding, M. Nottingham, D. Orchard, J. Gregorio und M. Hadley, „URI Template," IETF Secretariat, RFC 6570, März 2012.

A. Ergebnisse der Performancemessungen

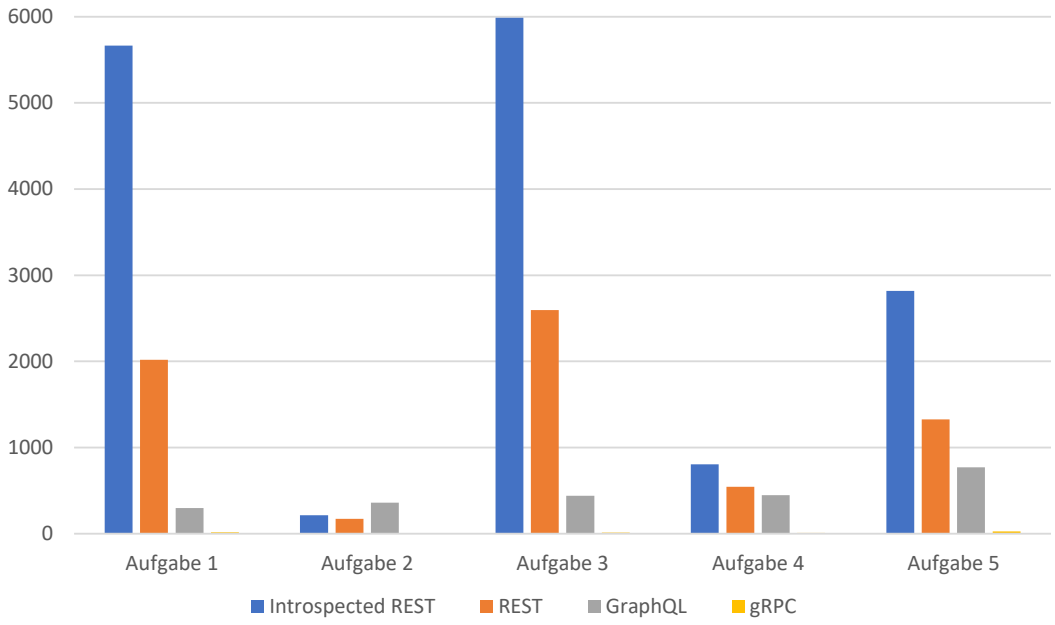

Abbildung A.1.: Gesamtzahl gesendeter Bytes je Aufgabe

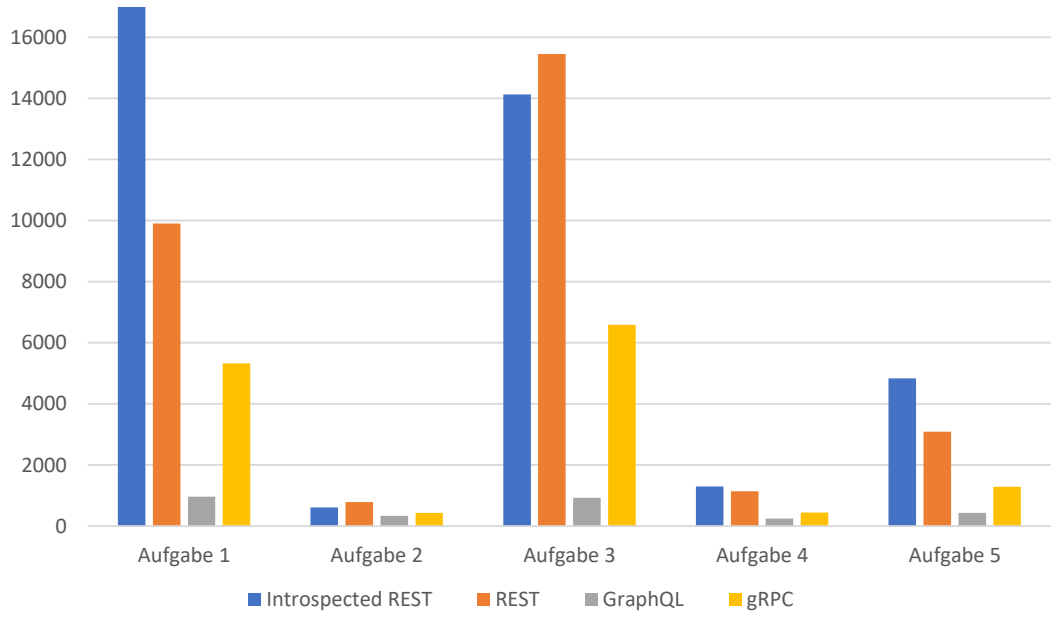

Abbildung A.2.: Gesamtzahl empfangener Bytes je Aufgabe

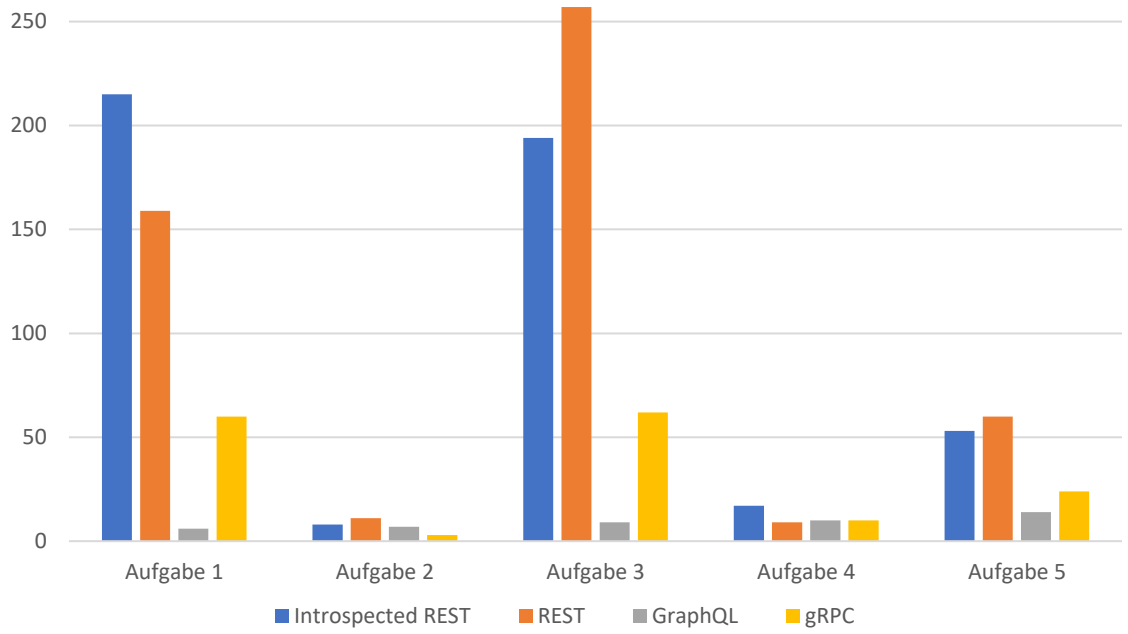

Abbildung A.3.: Antwortzeiten je Aufgabe in ms mit einem Client und Bandbreite 34KB/s Upload, 58KB/s Download

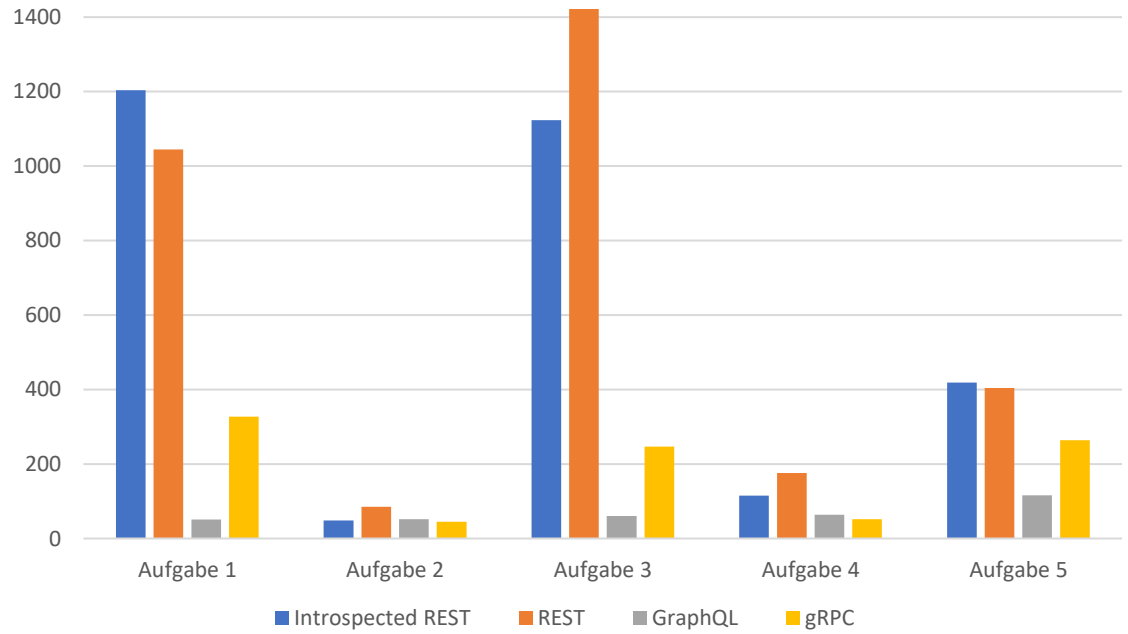

Abbildung A.4.: Antwortzeiten je Aufgabe in ms mit vier Clients und Bandbreite 34KB/s Upload, 58KB/s Download

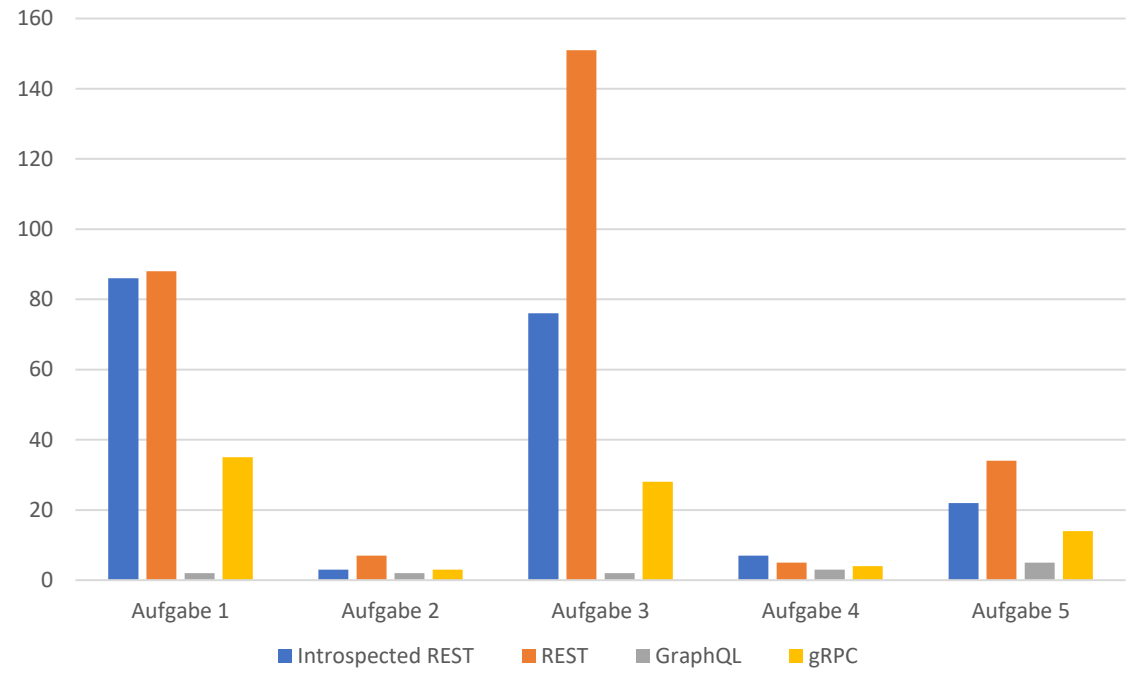

Abbildung A.5.: Antwortzeiten je Aufgabe in ms mit einem Client und Bandbreite 100KB/s Upload, 100KB/s Download

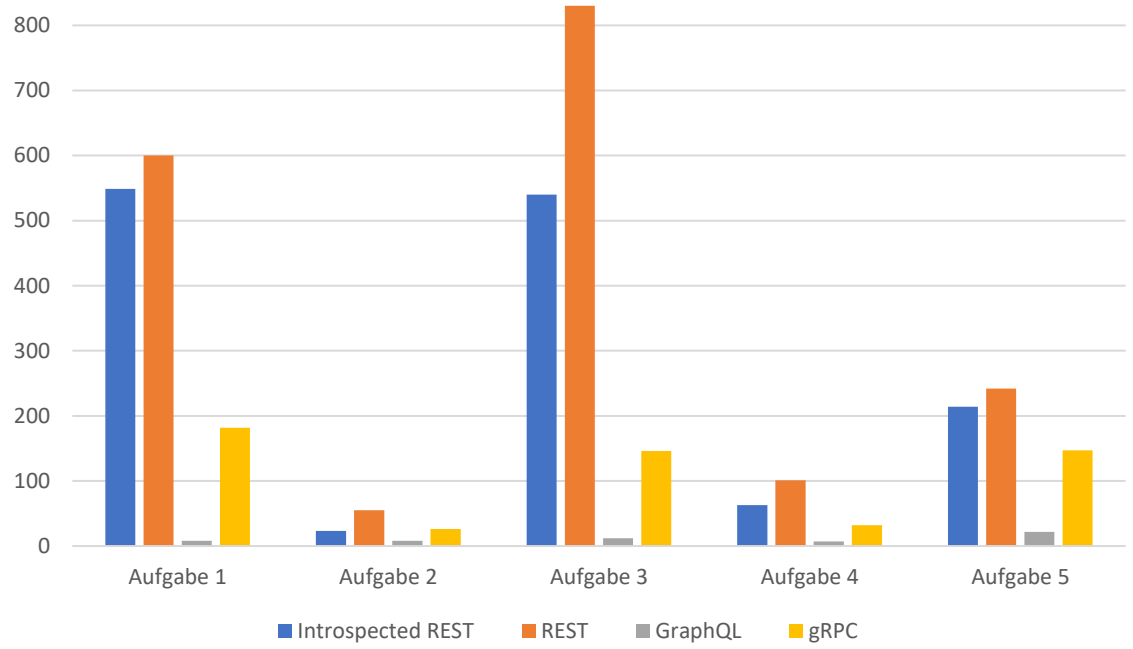

Abbildung A.6.: Antwortzeiten je Aufgabe in ms mit vier Clients und Bandbreite 100KB/s Upload, 100KB/s Download

B. Codebeispiele

```
{
    "_links": {
        "self": { "href": "/orders" },
        "next": { "href": "/orders?page=2" },
        "find": { "href": "/orders{?id}", "templated": true }
    },
    "_embedded": {
        "orders": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "basket": { "href": "/baskets/98712" },
                "customer": { "href": "/customers/7809" }
            },
            "total": 30.00,
            "currency": "USD",
            "status": "shipped",
        },{
            "_links": {
                "self": { "href": "/orders/124" },
                "basket": { "href": "/baskets/97213" },
                "customer": { "href": "/customers/12369" }
            },
            "total": 20.00,
            "currency": "USD",
            "status": "processing"
        }]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20
}
```

Listing B.1.: Payload eines HTTP-Response im HAL-Format

```
{
    "data": {
        "notes": [
            { "id": 1, "title": "Rezept" },
            { "id": 2, "title": "To-Do-Liste" },
        ]
    },
    "meta": {
        "offset-pagination": {
            "page": 1,
            "size": 2
        }
    }
}
```

Listing B.2.: Struktur des Container-Mediatypes

```
{
    "data": {
        "content": {
            "json-home": {
                "description": "Microtype for JSON Home Documents"
            },
            "json": {
                "description": "Microtype for plain JSON"
            }
        },
        "runtime": {
            "offset-pagination": {
                "category": "pagination",
                "documentation": "https://example.com/docs/offset-pag"
            }
        },
        "introspection": {
            "json-hyper-schema": {
                "configuration": {
                    "includeSelf": {
                        "possibleValues": [true, false],
                        "default": true
                    },
                    "includeValidationSchema": {
                        "possibleValues": [true, false],
                        "default": true
                    }
                }
            }
        }
    },
    "meta": {}
}
```


Listing B.3.: Introspection-Overview-Response mit Informationen über verschiedene Microtypes

Eidesstaatliche Erklärung

Ich bestätige hiermit, dass ich die vorliegende Bachelorarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Gutachter der Arbeit. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Leipzig, 18. September 2020

_____

Florian Gerlinghoff