

Eignung von GraphQL im Vergleich zu REST in verschiedenen Use Cases

Alexander Wolf

Abstract - In den letzten Jahrzehnten ist die Menge der verarbeiteten und versendeten Daten bei API-Abfragen immer weiter gestiegen. Mittlerweile werden bei der Verwendung von REST-APIs sowohl oftmals zu viele Daten als Antwort gesendet (*overfetching*), als auch vermehrt zu wenige Daten versendet (*underfetching*), was dazu führt, dass ein weiterer Request abgesendet werden muss. Diese beiden Phänomene zu vermeiden, würde sowohl Internet-Traffic reduzieren als auch unnötige Datenverarbeitung minimieren. GraphQL hat sich genau dies zum Ziel gemacht und versucht, den Benutzer entscheiden zu lassen, welche Daten in der Antwort enthalten sein sollen. Im Gegensatz zu REST kann es bei GraphQL jedoch zu Performanceeinbrüchen kommen. Es existiert also eine Grenze bei der, trotz *over-* und *underfetching*, die schiere Geschwindigkeit von REST den von GraphQL gebotenen Vorteilen zu bevorzugen ist. Dieses Paper soll also die Frage beantworten, in welchen Use Cases eine Architektur der anderen zu bevorzugen ist. Um herauszufinden, welche Szenarien in der Literatur bereits genauer untersucht und von Experten bewertet wurden, wurde eine detaillierte Literaturanalyse durchgeführt. Das Ergebnis dieser Analyse zeigte eine leichte Präferenz für GraphQL gegenüber REST, jedoch mit zwei konkreten Ausnahmen.

Index Terms— REST, GraphQL, web services, API, use cases.

I. EINLEITUNG

DAS REST (Representational State Transfer) Protokoll ist gegenwärtig der defacto Standard für die Erstellung von Web-APIs [1]. Jedoch birgt das Protokoll einige umstrittenen Design-Probleme, welche zu *over-* und *underfetching* bei HTTP Requests führen können [2]. Diese Probleme will das im Jahr 2012 von Facebook eingeführte GraphQL lösen [3]. Es verfolgt ein Prinzip, in dem Benutzer*innen genau definieren können, welche Attribute in der Antwort der API enthalten sein sollen [4]. Zusätzlich hat es den Vorteil, dass die gesamte API auf einer einzigen URL abgebildet werden kann. Zu sogenannten *Roundtrips*, welche in der Verwendung von REST üblich sind, kann es somit mit GraphQL nicht kommen. Nichtsdestotrotz schneidet GraphQL im Vergleich mit REST nicht in allen Punkten besser ab. Wie auch im Kapitel 4 später beschrieben wird, lässt die Performance, vor allem bei einer hohen Anzahl an Requests, zu wünschen übrig.

Es existiert in der Literatur keine allgemeine Übersicht, die beschreibt, welche Architektur der anderen in bestimmten Szenarien überlegen ist. Ein korrekter Einsatz der Technologien wäre von Vorteil, sowohl für Benutzer*innen, als auch Betreiber*innen von Web-APIs. Während GraphQL eine Verkleinerung der versendeten JSON-Objekte von bis zu 99% (in Bytes) ermöglicht [5], gelingt REST ein schnelleres Verarbeiten

von Requests, unter der Voraussetzung das Design der API ist exakt auf die einkommenden Anfragen abgestimmt. Das Ziel dieses Papers ist also zu erläutern, in welchen Situationen es sich lohnt GraphQL einzusetzen und in welchen die Geschwindigkeit von REST zu bevorzugen ist. Die Forschungsfrage dieses Papers lautet nun wie folgt:

RQ1: In welchen Szenarien ist GraphQL gegenüber REST als Architektur für APIs, dem aktuellen Stand der Literatur nach, zu bevorzugen?

II. STRUKTUR

Im nachfolgenden Kapitel wird genauer auf die verwendete Methodik und Suchstrategie eingegangen, welche die Basis für die Literaturrecherche bildeten. In Kapitel 4 werden die unterschiedlichen Funktionsweisen von REST und GraphQL anhand von Beispielen erläutert. Kapitel 5 behandelt eine Gegenüberstellung der beiden Architekturstile anhand einer Reihe essenzieller Metriken. Nachfolgend wird in Kapitel 6 beschrieben, in welchen Use Cases Experten die Verwendung von REST oder GraphQL bevorzugen. Darauf aufbauend wird in Kapitel 7 „Ergebnisse“ ein zusammenfassender Überblick geboten, der die Eignung der Technologien in bestimmten Szenarien einordnen soll. Das Résumé gibt schlussendlich einen Ausblick über offene Forschungsfragen und hebt entdeckte Inkonsistenzen zwischen unterschiedlichen Quellen hervor.

III. METHODIK

Dieses Paper ist aufgebaut auf einer umfassenden Analyse der Literatur bezüglich eines Vergleichs zwischen der REST-Architektur und GraphQL anhand konkreter Szenarien. Darüber hinaus wurde sich im Detail mit den technologischen Aspekten der beiden Architekturen beschäftigt und jeweils ein API-Service mit REST und GraphQL umgesetzt, um ein besseres Verständnis der Technologien zu erlangen. Auch Literatur bezüglich allgemeiner Kriterien, wie beispielsweise Performance und Sicherheit, wurde in die Recherche für dieses Paper miteinbezogen.

Im ersten Schritt der Literatursuche wurden alle Datenbanken nach einer Kombination aus den Suchtermini *REST* und *GraphQL* durchforstet. Je nach Genauigkeit des Ergebnisses und der Anzahl der gefundenen Literatur wurde die Suchstrategie weiter angepasst. Bei den stärker eingegrenzten Suchen wurden die Suchtermini *comparison* und *architecture* hinzugefügt. Die Literatursuche schloss von Beginn an Preview-Only Papers aus. In der Literaturdatenbank ACM mussten die Begriffe *blockchain* und *teaching* aus der Suche ausgeschlossen sowie das Publikationsdatum auf 2019-2023 eingeschränkt werden. In den Literaturdatenbanken ACM, IEEE Xplore und Springer konnten passenden Ergebnisse gefunden werden, Elsevier hingegen lieferte keine Papers, die letztendlich in die Literaturliste aufgenommen wurden.

IV. FUNKTIONSWEISE VON REST UND GRAPHQL

A. REST

Wird REST als Architekturstil für die Erstellung einer API verwendet so können Ressourcen durch den Aufruf von URLs abgefragt, verändert, erstellt oder gelöscht werden. Um den REST-Richtlinien zu entsprechen, sollte dabei jeweils die korrekte HTTP-Methode, beispielsweise GET oder POST, verwendet werden. REST ist ein zustandsloses Protokoll, was bedeutet, dass weder der Client noch der Server den Zustand des anderen kennen [6]. Darüber hinaus verwendet REST das extensive HTTP-Caching. Ist die Antwort auf eine Abfrage bereits in einem lokalen oder proxy Cache gespeichert, braucht der eigentliche REST-Server nicht mehr weiter angefragt werden.

```
1 Request:
2 GET http://geodb.com/city/29381
3 Accept: application/json
4
5 Response:
6 Status Code: 200
7 Content-type: application/json
8 Body: {
9   "id": 29381,
10  "cityname": "Linz",
11  "population": 210000,
12  "region": "Oberösterreich",
13  "postal-codes": [
14    {
15      "name": "Urfahr",
16      "postal-code": 4040
17    },
18    {
19      "name": "Innenstadt",
20      "postal-code": 4020
21    },
22    {
23      "name": "Neue Heimat",
24      "postal-code": 4030
25    }
26  ]
27 }
28
```

Abbildung 1 Beispielhafter REST-API Aufruf

Wie in Abbildung 1 zu sehen ist, wird die ID der gesuchten Stadt über die URL mitgegeben. Die Antwort der API beinhaltet die vom Entwickler vorgegebenen Felder. Im Falle, dass eines oder mehrere der Felder, beispielsweise die *region*, nicht benötigt werden, gibt es für den Benutzer keine Möglichkeit, diese schon im Vorhinein aus der Antwort auszuschließen. Dieses Problem wird auch *overfetching* genannt. Im Gegensatz dazu könnte den Benutzer, zusätzlich zu den gesendeten Informationen, eine Liste der umliegenden Gemeinden interessieren. Da diese Informationen nicht in der originalen Antwort enthalten ist, muss ein weiterer Request abgesendet werden; man spricht hier von *underfetching*.

B. GraphQL

Im Gegensatz zu REST wird ein API-Zugriff über eine einzige URL ermöglicht. Die wirkliche Abfrage wird über den Body des HTTP-Requests mitgegeben. Wichtig zu erwähnen ist, dass es Best Practice ist, den HTTP-Request als POST Request abzusenden, auch wenn möglicherweise nur Daten abgefragt und nicht verändert werden [4]. Auch wenn es möglich wäre, einem GET Request einen Body anzuhängen, ist dies laut HTTP-Spezifikation nicht empfohlen [7]. Die Syntax der Abfrage ist für simple Abfragen leicht verständlich, vorausgesetzt, man besitzt bereits Kenntnis von Web-Technologien. Im Gegensatz zu REST und den HTTP-Verben gibt es für GraphQL nur zwei Arten von Operationen, Mutationen und Queries. Im GraphQL Schema müssen sowohl für Mutationen als auch

für Queries Typen festgelegt sein. Aufgrund der Tatsache, dass laut Spezifikation GraphQL POST Requests verwendet, um mit der API zu interagieren, gibt es keinen impliziten Caching Mechanismus, da HTTP/1.1 kein Caching von POST Requests unterstützt [7].

```

1 Request:
2   POST http://geodb.com/graphql
3   Accept: application/json
4   Body: {
5     "query": "
6       query {
7         city(id:29381) {
8           cityname
9           population
10          postal-codes(first: 2) {
11            name
12          }
13        }
14      }
15    "
16  }
17
18 Reponse:
19   Status Code: 200
20   Content-type: application/json
21   Body: {
22     "cityname": "Linz",
23     "population": 210000,
24     "postal-codes": [
25       {
26         "name": "Urfahr
27       },
28       {
29         "name": "Innenstadt"
30       }
31     ]
  }

```

Abbildung 2 Beispielhafter GraphQL-API Aufruf

Abbildung 2 macht den Unterschied zu REST deutlich ersichtlich. Anstatt nur über die URL eine ID abzufragen, wird hier eine gesamte Abfrage im Body des Requests mitgesendet. Mit Hilfe spezifischer Syntax kann der Benutzer angeben, welche Felder in der Antwort enthalten sein sollen. Zusätzlich können in der Antwort enthaltene Arrays auch in ihrer Länge und Inhalt angepasst werden. In Abbildung 2 wird die gesuchte ID im *city* Attribut der Query mitgegeben. Weiter wird spezifisch nur nach den ersten beiden Elementen der Liste *postal-codes* gefragt, mit der zusätzlichen Einschränkung, dass jeweils nur der Name in der Antwort enthalten sein soll.

V. GEGENÜBERSTELLUNG DER BEIDEN ARCHITEKTUREN

Es ist wichtig zu erwähnen, dass keine der beiden Architekturen der anderen in ihren Funktionalitäten überlegen ist. Jede Operation kann für beide Systeme mit gleichem Ergebnis durchgeführt werden.

A. Performance

Performanceunterschiede zwischen den beiden Architekturen sind eindeutig gegeben. Es wurde mehrfach gezeigt, dass REST im Durchschnitt und vor allem ab einer

gewissen Menge gleichzeitiger Requests bzw. Benutzer*innen deutlich schneller Anfragen beantwortet als GraphQL [2], [8], [9]. Bei niedriger Anzahl von zu verarbeitenden Requests ist der Unterschied in der Performance nicht signifikant [1]. Natürlich muss bedacht werden, dass die zitierten Studien von einer perfekt ausgelegten REST-API ausgehen und nicht in Betracht ziehen, dass Benutzer*innen möglicherweise mehr als einen Request absenden müssen, um an alle benötigten Daten zu gelangen. GraphQL braucht länger, um die gewünschten Daten zu liefern, jedoch wird garantiert, dass kein zweiter Request gesendet werden muss.

B. Erlern- und Wartbarkeit

Es muss bedacht werden, dass nicht nur die Performance eine Rolle spielt, wenn es darum geht, Technologien in bestehenden kritischen Systemen zu integrieren. Auch die leichte Erlern- und Anpassbarkeit muss gegeben sein, um aus unternehmerischer Perspektive wirtschaftlich korrekt zu handeln. Brito [10] untersuchte die durchschnittliche Entwicklungsdauer einer einfachen API anhand von 22 Student*innen mit teilweise vorhandener Erfahrung. Eine überwiegende Mehrheit der Student*innen empfand GraphQL als leichter handhabbar als REST. Darüber hinaus war auch die durchschnittliche Dauer der Entwicklung mit GraphQL, inklusive des Schreibens von einigen Queries, signifikant niedriger.

Nicht gänzlich widersprüchlich steht diesem Ergebnis das Resultat von [1] gegenüber. Neben einer quantitativen Studie zur Antwortgeschwindigkeit von GraphQL und REST beinhaltet [1] auch einen qualitativen Teil, in welchem Arbeitnehmer*innen von GitHub ihre professionelle Meinung zu GraphQL preisgeben. Im Gegensatz zu [10] hatten die Mitarbeiter*innen eine überwiegend negative Einstellung gegenüber GraphQL. Nach Meinung der Mitarbeiter*innen stellt die Zeit, welche man investieren muss, um Anforderungen effektiv erfüllen zu können, eine Herausforderung dar. Noch dazu beklagen einige der Befragten, dass es zu wenige Entwickler*innen gibt, die ein tiefgehendes Verständnis von GraphQL haben und deswegen ein Migrieren von bestehenden REST-Systemen zu GraphQL schwierig ist.

Dieser drastische Unterschied in der Wahrnehmung der Erlernbarkeit könnte durch einen Komplexitätsunterschied erklärt werden. GraphQL in GitHubs Systeme zu integrieren erfordert höchstwahrscheinlich weitaus komplexere Lösungen als die grundlegenden APIs, die in [10] implementiert wurden. Das lässt jedenfalls die Frage offen, ob GraphQL überhaupt zu empfehlen ist, wenn Entwickler*innen großer, skalierten Systeme davon abraten.

Ein wichtiger, bisher noch nicht erwähnter, Aspekt bezüglich der Wartbarkeit von REST-APIs ist die notwendige und oftmals komplizierte Versionierung. Wie bei Software im Allgemeinen üblich, wird von den Entwicklern*innen versucht, Rückwärtskompatibilität zu erreichen. Für REST-APIs bedeutet das, dass ab Einführung der API bei gleicher Abfrage das exakt gleiche Ergebnis geliefert werden muss. Änderungen in der Struktur der Antworten können somit nicht durchgeführt werden, ohne die auf die API angewiesenen Systeme zu beeinträchtigen.

Dieses Problem hat GraphQL nicht, da die Struktur der Antwort im Prinzip die Benutzer*innen selbst vorgeben. Natürlich muss trotzdem Sorge dafür getragen werden, dass stets dieselben Informationen abrufbar sind [1].

C. Sicherheit

Während durch die jahrelange Dominanz der REST-Architektur die möglichen Sicherheitsprobleme, wie beispielsweise *Code Injection* oder *Replay Attacks*, den Entwickler*innen wohl bekannt sind, gibt es noch zu wenig Expertise in der Verwendung von GraphQL, was dazu führt, dass oftmals Sicherheitslücken in den Schemen der APIs übersehen werden [11]. Insbesondere *Denial of Service-Angriffe* sind vermehrt bei undurchdacht implementierten GraphQL-APIs möglich.

D. Portierbarkeit

Wittern *et al.* [12] zeigt, dass es eine Möglichkeit gibt GraphQL Schemen aus bestehenden REST-APIs zu generieren und somit beide Architekturen für Benutzer*innen anzubieten. Das Paper weist jedoch darauf hin, dass nicht alle APIs fehlerfrei abgebildet werden können und man eventuell manuell Anpassungen durchführen muss. Laut derzeitigem Stand der Literatur gibt es keine Option, eine GraphQL-API zu einer REST-API zu portieren.

E. Testbarkeit

Das Testen von REST-APIs ist ein gründlich erforschtes Thema. Sowohl die automatische Erstellung von Tests als auch die automatisierte Durchführung stellen kein Problem für skalierte REST-APIs dar [13]. Im Gegensatz dazu stellt sich das Testen von GraphQL-Applikationen als eine nicht triviale Aufgabe heraus. Das Problem liegt in der verhältnismäßig geringen Anzahl an Frameworks, die ein implementiertes GraphQL Schema automatisch auf Inkonsistenzen überprüfen können. Der Fortschritt liegt hier noch hinter den etablierten REST-Test-Frameworks. Diese Imbalance sollte jedoch in den nächsten Jahren und nach weiterer Etablierung von GraphQL ausgeglichen werden. Vargas *et al.* [14] stellt zudem als Alternative eine neuartige Testmethodik vor, mit der GraphQL Schemen gründlicher getestet werden können als mit regulären Testmethoden.

F. Netzwerkbelastung

Der große Vorteil von GraphQL liegt in der Tatsache, dass garantiert werden kann, dass nur ein Request gesendet werden muss, um an alle benötigten Daten zu gelangen. Dies hat zur Folge, dass nur ein Bruchteil von Daten bei der Verwendung von GraphQL gesendet werden muss im Vergleich mit der Verwendung von REST, wie [5] auch belegt. Interessant wären hierzu Daten, wie GraphQL im Vergleich zu REST das Netzwerk belastet und somit den Stromverbrauch mindert. Natürlich müsste der lokale Stromverbrauch am Server auch miteinkalkuliert werden, weil davon auszugehen ist, dass GraphQL mehr Berechnungen durchführt und somit mehr CPU-Zeit als REST in Anspruch nimmt. Bei einem signifikanten Unterschied könnte man im Sinne der Umweltverträglichkeit argumentieren, dass es ratsam wäre,

GraphQL zu wählen. Daten dazu gibt es in der Literatur noch keine, hier wäre mehr Forschung wünschenswert.

VI. USE CASES FÜR DIE BEIDEN ARCHITEKTUREN

Nun bleibt die Frage, unter welchen Rahmenbedingungen es zu empfehlen ist, GraphQL oder REST zu verwenden. Das Risiko, dass eine der beiden Technologien in der Zukunft nicht mehr relevant sein sollte und es damit zu einem Mangel an Entwickler*innen kommen könnte, besteht nicht, wenn man die Trends der letzten Jahre betrachtet.

In einem Teil der Befragung aus Vadlamani *et al.* [1] wurden die Mitarbeiter*innen gefragt, welche Architektur sie in gewissen Szenarien bevorzugen würden. Leider wird in der Tabelle der Ergebnisse nur aufgezeigt welche Architektur die Mehrheit der Befragten gewählt hatten, mit keiner Indikation, wie groß die Mehrheit in dem jeweiligen Szenario wirklich gewesen ist. Ungeachtet dessen zeigen die Ergebnisse, dass REST präferiert wird, wenn es um kritische Echtzeit-APIs geht als auch für kleinere interne und öffentliche APIs. Der Grund, warum genau auch bei kritischen APIs REST bevorzugt wird, wird nicht weiter erläutert. Eine Erläuterung könnten die Sicherheitsprobleme sein, welche bereits in diesem Paper angesprochen wurden.

GraphQL hingegen wird von einer Mehrheit der Befragten bevorzugt, wenn das betroffene Backend von einer Vielzahl unterschiedlicher interner Clients benutzt wird. Begründet werden kann dieses Ergebnis durch die zuvor bereits angesprochene Tatsache, dass die Versionierung von APIs mit GraphQL weitaus einfacher ist als mit REST.

Auch bei mittelgroßen und großen Projekten wird von den Mitarbeiter*innen GraphQL über REST bevorzugt. Dieses Ergebnis verwundert etwas, da in einigen Ausschnitten des Papers Befragte sich über die Komplexität und Geschwindigkeit von GraphQL beschwerten. Man kann nur vermuten, dass die Entwickler die strikte Typisierung von GraphQL präferieren.

Nicht näher im Detail angesprochen wurde die Verfügbarkeit von Entwickler*innen mit tiefgehendem Wissen in den Architekturen. Die Tatsache, dass Entwickler*innen im Durchschnitt weit mehr Erfahrung mit REST als mit GraphQL haben, muss in die Entscheidung, welche Architektur in einem Projekt zu verwenden ist, miteinbezogen werden. Vor allem, wenn es sich um ein Projekt handelt, welches erst in den Startlöchern steht, ist man, meines Erachtens nach, besser beraten REST als API-Architektur zu wählen.

Wie schon im Kapitel über die Performance näher beleuchtet, ist ein deutlicher Unterschied in der Reaktionszeit bei der Verwendung in Projekten mit viel Traffic gegeben. Laut Heredia *et al.* [8] besteht schon bei 500 gleichzeitigen Nutzern ein signifikanter Unterschied in der Reaktionszeit der Systeme zugunsten von REST. Eine andere Wahl als REST wäre in einem Projekt, in welchem die schnelle Verarbeitung eines Requests essenziell ist, nicht empfehlenswert. Spielt Performance hingegen keine Rolle, sprechen mehr Faktoren für die Wahl von GraphQL. Als Bonus besteht die Möglichkeit, dass man durch die

Wahl von GraphQL sogar auf indirektem Wege die CO₂ Emissionen seiner Applikation minimiert, wie im Kapitel über die Netzwerkbelastung bereits angesprochen.

Angenommen Ressourcen, finanzieller und zeitlicher Art, spielen keine Rolle, spricht nichts dagegen, beide Architekturen anzubieten und über einen gewissen Zeitraum zu betrachten, für welche sich die Benutzer*innen vermehrt entscheiden. Im Zuge dessen bietet es sich auch an, die von Wittern *et al.* [12] vorgestellte Technologie zum Generieren von Wrapper für REST-APIs zu verwenden. Dies dürfte den insgesamt Zeitaufwand der Entwicklung der API-Schnittstellen verringern.

VII. ERGEBNISSE

Aus den diversen Papers lässt sich schließen, dass im generellen GraphQL über REST bevorzugt wird. Ausnahmen bilden hier das Szenario einer sicherheitskritischen Echtzeit-Applikation und eine API, bei welcher Performance an oberster Stelle steht. In beiden Fällen sollte REST verwendet werden. Es gibt Grenzfälle, in denen die Entscheidung auf beide Architekturen entfallen kann. Beispielsweise eine vollkommen neu zu entwickelnde API, deren Entwicklerbasis keine Kenntnisse von GraphQL hat, jedoch Erfahrung mit der REST-Architektur besitzt. Hier ist die Entscheidung lediglich an die verfügbaren Zeitkapazitäten der Entwickler*innen, welche GraphQL erst erlernen müssten, angeknüpft. Unklar bleibt, auf welche Technologie eine Entscheidung entfallen sollte, falls die Skalierbarkeit eine zentrale Rolle spielt. Bestehen aus Sicht des Managements keine besonderen Anforderungen an das System, sollte es den Mitarbeitern freistehen, eine der beiden Technologien zu wählen.

VIII. RÉSUMÉ

In diesem Paper wurde eine kurze Einführung zu REST und GraphQL gegeben und die Funktionsweisen der beiden Architekturen erläutert. Daraufhin wurden die beiden Systeme anhand von relevanten und kritischen Aspekten gegenübergestellt. Zu guter Letzt wurde eine Empfehlung aufgrund der derzeit verfügbaren Literatur gegeben, welche Architektur bei welchen Gegebenheiten zu verwenden ist. Besteht ein hoher Anspruch an die Performance oder die Sicherheit, führt kein Weg an der REST-Architektur vorbei. Spielen beiden Aspekte in der betroffenen Applikation eine sekundäre Rolle, ist GraphQL die bessere Wahl. Diese Empfehlung betrachtet die Wahl aus Entwickler-Perspektive. Für die Frage, welche der beiden Architekturen für Benutzer*innen attraktiver ist, existiert derzeit keine Literatur. Wie schon im Kapitel über *Netzwerkbelastung* erwähnt, wäre eine Studie über den ökologischen Fußabdruck der Architekturen interessant. Die Information, welche der beiden Architekturen den größeren ökologischen Impact hat, könnte man daraufhin auch in die Entscheidung miteinfließen lassen. Vor allem wenn bestehende Sicherheitsprobleme behoben werden, könnte GraphQL sich als potenziell grüne Alternative einen Namen machen. Auch die empfundene Leichtigkeit der

Handhabung von GraphQL könnte, aufgrund der unterschiedlichen Ergebnisse aus [1] und [10], näher untersucht werden.

REFERENCES

- [1] S. L. Vadlamani, B. Emdon, J. Arts, und O. Baysal, „Can GraphQL Replace REST? A Study of Their Efficiency and Viability“, in *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, Juni 2021, S. 10–17. doi: 10.1109/SER-IP52554.2021.00009.
- [2] A. Lawi, B. L. E. Panggabean, und T. Yoshida, „Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System“. 22. September 2021. doi: 10.20944/preprints202109.0386.v1.
- [3] L. Byron, „GraphQL: A data query language“, *Engineering at Meta*, 14. September 2015. <https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/> (zugegriffen 21. Dezember 2022).
- [4] „GraphQL Specification“. <https://spec.graphql.org/June2018/> (zugegriffen 26. Dezember 2023).
- [5] G. Brito, T. Mombach, und M. T. Valente, „Migrating to GraphQL: A Practical Assessment“, in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2019, S. 140–150. doi: 10.1109/SANER.2019.8667986.
- [6] R. Fielding, „Architectural Styles and the Design of Network-based Software Architectures“; Doctoral dissertation“, 2000. Zugegriffen: 27. Dezember 2022. [Online]. Verfügbar unter: <https://www.semanticscholar.org/paper/Architectural-Styles-and-the-Design-of-Software-Fielding/49fc9782483bc311c2bd1b902dfb32bcd99ff2d3>
- [7] H. Nielsen *u. a.*, „Hypertext Transfer Protocol – HTTP/1.1 - Specification“, Internet Engineering Task Force, Request for Comments RFC 2616, Juni 1999. doi: 10.17487/RFC2616.
- [8] J. Sayago Heredia, E. Flores-García, und A. R. Solano, „Comparative Analysis Between Standards Oriented to Web Services: SOAP, REST and GRAPHQL“, in *Applied Technologies*, Cham, 2020, S. 286–300. doi: 10.1007/978-3-030-42517-3_22.
- [9] N. Vohra und I. B. Kerthyayana Manuaba, „Implementation of REST API vs GraphQL in Microservice Architecture“, in *2022 International Conference on Information Management and Technology (ICIMTech)*, Aug. 2022, S. 45–50. doi: 10.1109/ICIMTech55957.2022.9915098.
- [10] G. Brito und M. T. Valente, „REST vs GraphQL: A Controlled Experiment“. arXiv, 10. März 2020. doi: 10.48550/arXiv.2003.04761.
- [11] E. Wittern, A. Cha, J. C. Davis, G. Baudart, und L. Mandel, „An Empirical Study of GraphQL Schemas“. arXiv, 30. Juli 2019. doi: 10.48550/arXiv.1907.13012.

- [12] E. Wittern, A. Cha, und J. A. Laredo, „Generating GraphQL-Wrappers for REST(-like) APIs“, Bd. 10845, 2018, S. 65–83. doi: 10.1007/978-3-319-91662-0_5.
- [13] M. Kim, Q. Xin, S. Sinha, und A. Orso, „Automated test generation for REST APIs: no time to rest yet“, in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, Juli 2022, S. 289–301. doi: 10.1145/3533767.3534401.
- [14] D. M. Vargas u. a., „Deviation Testing: A Test Case Generation Technique for GraphQL APIs“, 2018. Zugegriffen: 3. Jänner 2023. [Online]. Verfügbar unter:
<https://www.semanticscholar.org/paper/Deviation-Testing%3A-A-Test-Case-Generation-Technique-Vargas-Blanco/bd8872cb86b7cf94821ccd78ee9abb0a7c4c17d>