



Dominik Kress

GraphQL

Eine Einführung in APIs mit GraphQL

dpunkt.verlag



Dominik Kress ist Software Engineer mit Heimat im E-Commerce. In seiner langjährigen Arbeit bei der größten Retail-Gruppe Europas hilft er bei der Modernisierung der internationalen Onlineshop-Systeme. Bei der Transformation von einem klassischen On-Premise-Monolithen zu einer Cloud-basierten Self-Contained-Systems-Architektur entwickelte und vertiefte sich seine Liebe zum Thema APIs. Sowohl im Umfeld seiner Arbeit als auch in privaten Projekten probiert er sich leidenschaftlich gerne an neuen Technologien, Spezifikationen und Methodiken. Daher ist für ihn der Austausch von Wissen und Erfahrungen auf Veranstaltungen und Konferenzen, die er auch gerne selbst nebenbei organisiert, besonders wichtig.

Papier
plus⁺
PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus⁺:

www.dpunkt.plus

Dominik Kress

GraphQL

Eine Einführung in APIs mit GraphQL



dpunkt.verlag

Dominik Kress

Lektorat: Melanie Andrisek

Copy-Editing: Claudia Lötschert, www.richtiger-text.de

Satz: Da-TeX Gerd Blumenstein, Leipzig, www.da-tex.de

Herstellung: Stefanie Weidner

Umschlaggestaltung: Helmut Kraus, www.exclam.de

Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN:

Print 978-3-86490-784-5

PDF 978-3-96910-121-6

ePub 978-3-96910-122-3

mobi 978-3-96910-123-0

1. Auflage 2021

Copyright © 2021 dpunkt.verlag GmbH

Wieblinger Weg 17

69123 Heidelberg

Hinweis:

Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.



Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Vorwort

Schon seit ich denken kann, fasziniert mich das Internet. Jemand am anderen Ende der Welt tippt etwas in seinen Computer, und ich bekomme diese Eingabe Tausende Kilometer entfernt zu sehen. Wie das genau funktioniert, war für mich schon immer eine der spannendsten Fragen.

Als ich dann irgendwann mit meinem Studium der Softwareentwicklung begann, konnte ich die Vorlesung »Verteilte Systeme« kaum abwarten. Doch als es endlich so weit war, kam die Ernüchterung: Das Internet bestand laut meinem Professor aus komplexen J2EE-Webservices, die gigantische XML-Strukturen über noch größere SOAP-Schnittstellen schieben.

Das hatte ich mir alles aufregender vorgestellt. Dabei liegt das Studium mit meinem Abschluss 2017 nicht mal in so ferner Vergangenheit. Doch gerade an Universitäten dreht sich die Zeit in einem ganz eigenen Rhythmus. Zum Glück half ein Hoffnungsschimmer, eigene Wege zu finden: eine Einführung in das Architekturmuster REST.

Diese eigenen Wege führten mich während meiner Bachelor-Arbeit über Schnittstellen in verteilten Systemen zu GraphQL. Da hatte ich es endlich: eine aufregend elegante Art, Applikationen – und damit Menschen – aus aller Welt zu verbinden.

Um diese Begeisterung zu teilen sowie anderen Anfängern bis fortgeschrittenen Webentwicklern zu zeigen, wie die Kommunikation zwischen Applikationen im Internet auch elegant funktionieren kann, habe ich dieses Buch geschrieben.

Für wen ist dieses Buch?

Dieses Buch ist für alle, die Interesse an Schnittstellen in verteilten Systemen und insbesondere an GraphQL haben. Die Motivation war, ein Grundlagenwerk zu schaffen, durch das man sowohl APIs im Allgemeinen als auch GraphQL im Speziellen verstehen lernt.

Gerade in den ersten beiden Kapiteln wird daher zuerst das generelle Konzept erklärt – unabhängig von der GraphQL-Spezifikation. Ver-

mittelt wird hier sowohl, was Schnittstellen in verteilten Systemen sind, als auch, worauf man bei ihrer Konzeption achten muss. Diese Kapitel richten sich ganz klar eher an Anfänger im Bereich der APIs.

Das Kapitel 1.4 über API-Technologien und -Spezifikationen im Vergleich mag auch für Fortgeschrittene interessant sein. Ansonsten ist für diese Zielgruppe ein Einstieg in Kapitel 3 ratsam.

Ab dort wird die API-Umsetzung anhand von GraphQL – und damit auch die Spezifikation selbst – ausgiebig vorgestellt. Neben den Möglichkeiten mit der Technologie werden auch Designempfehlungen und Implementierungsbeispiele sowohl mit JavaScript als auch mit Java vermittelt. Gerade für die letzten beiden Teile wird ein Grundlagenwissen in der Softwareentwicklung mit JavaScript und Java erwartet.

Was soll dieses Buch vermitteln?

Dieses Buch soll ein Grundlagenwissen schaffen. Ob man in der ersten Vorlesungsstunde des Kurses »Verteilte Systeme« eine modernere Alternative zu dem dort Vorgestellten sucht oder sich als fortgeschrittener Entwickler schlicht für GraphQL interessiert: Nach der Lektüre dieses Buches sollte sowohl verstanden sein, wie APIs mit GraphQL funktionieren, als auch, wie sie umgesetzt werden.

Die Implementierungsbeispiele der letzten beiden Kapitel dienen dabei als Boilerplate-Code, der gut zum Start eines eigenen Projekts mit GraphQL verwendet werden kann. Dessen Konzept zu verstehen und ihn weiterentwickeln zu können, ist das Ziel dieses Buches.

Inhaltsverzeichnis

Vorwort	v
1 API-Grundlagen	1
1.1 Was ist ein API?	1
1.2 Vorteile eines API	3
1.2.1 Flexibilität für Anbieter und Konsument	3
1.2.2 Einheitliches Design und Funktionen	4
1.2.3 Neue Geschäftsfelder	4
1.2.4 Innovationstreiber API	4
1.3 API: Die Definition	5
1.3.1 API-Vertrag	5
1.3.2 Die Akteure eines API	6
1.3.3 Release-Arten von APIs	7
1.4 Mögliche API-Technologien und -Spezifikationen	8
1.4.1 Geschichte der Remote Execution	8
1.4.2 RESTful HTTP	10
1.4.3 JSON:API	12
1.4.4 gRPC	15
1.4.5 GraphQL	18
1.4.6 Die Technologien im Vergleich	20
2 Von der Idee zur Umsetzung	21
2.1 API Value Chain	22
2.1.1 Geschäftsmodelle für private und öffentliche APIs ...	24
2.2 Release-Arten	27
2.2.1 Private APIs	27
2.2.2 Öffentliche APIs	28
2.3 Erste Schritte: Allgemeines Vorgehen	29
2.3.1 Use Cases identifizieren	29
2.3.2 Funktionale Anforderungen	31
2.3.3 Nicht-funktionale Anforderungen	32
2.3.4 Die gemeinsame Sprache	32
2.3.5 Gemeinsames Vokabular durch Schema.org erzeugen	33

3	Grundlagen der GraphQL-API	37
3.1	Das Graphen-Modell erzeugen	38
3.2	Abfragen mit GraphQL	40
3.2.1	Grundlegende Querys	40
3.2.2	Querys generell	41
3.2.3	Verschachtelte Querys	42
3.2.4	Parameter in Querys	43
3.2.5	Variablen in Querys	44
3.2.6	Aliases in Querys	46
3.2.7	Fragmentierte Querys	47
3.2.8	Direktiven in Querys	49
3.2.9	Inline-Fragmente in Querys	51
3.2.10	Metafelder in Querys	52
3.2.11	Mutationen: Datenmanipulation mit GraphQL	54
3.2.12	Subscriptions: GraphQL Message Streaming	55
3.3	Das GraphQL-Typ-System: Schemadefinition	57
3.3.1	Grundlegende Schemas	58
3.3.2	Skalar-Typen	60
3.3.3	Enumerations-Typen	60
3.3.4	Typ-Modifikatoren: Listen und Non-Null	61
3.3.5	Parameter	62
3.3.6	Input-Typen	63
3.3.7	Interfaces	65
3.3.8	Union-Typen	66
4	API-Design I: Rund ums Schema des API	67
4.1	Qualitätsmerkmale	67
4.2	Designempfehlungen	71
4.2.1	Schemadesign-Empfehlungen	72
4.2.2	Mutation-Designempfehlungen	77
4.3	HTTP: Netzwerk-Design	81
4.3.1	POST-Requests	81
4.3.2	GET-Requests	82
4.3.3	Responses	83
4.4	Pagination	84
4.4.1	Splicing	84
4.4.2	Offset-basierte Pagination	85
4.4.3	Cursor-basierte Pagination	85
4.4.4	Edges und Connections	86
4.5	Fehlermanagement	88
4.5.1	Application Errors	89
4.5.2	Type und Validation Errors	89
4.5.3	Fehler mit partiellen Ergebnissen	90
4.5.4	Fehler ohne Teilergebnisse	91

5	API-Design II: Die Landschaft um das API	93
5.1	Autorisierung	93
5.1.1	GraphQLs Probleme mit Autorisierung	93
5.1.2	Autorisierung auf Ebene der Geschäftslogik	94
5.2	Dokumentation	95
5.2.1	Statische Dokumentation	96
5.2.2	Dynamische Dokumentation	98
5.3	Versionierung	100
5.3.1	GraphQLs Evolution im Beispiel	101
5.4	Monitoring und Instrumentation	102
5.4.1	Feingranulares Monitoring	102
5.4.2	Verstehen, wie das API genutzt wird	103
5.5	Performanzoptimierung: Caching und Batching	103
5.5.1	Das 1+n-Problem	104
5.5.2	DataLoader	105
5.5.3	CDN-Caching	106
5.5.4	Clientseitiges Caching	107
6	Implementierung mit Node I: Das erste Schema	109
6.1	Use Case	109
6.2	Initiales Aufsetzen des Projekts	110
6.2.1	Das Node.js-Projekt aufsetzen	110
6.2.2	Den GraphQL-Server mit Apollo aufsetzen	111
6.3	Das initiale Schema aufsetzen	113
6.3.1	Parameter und erste Resolver-Logik	114
6.3.2	Feld-Level-Resolver und Interfaces	116
6.3.3	Interfaces und Filter für IDs	118
6.3.4	Typrelationen	122
7	Implementierung mit Node II: Erweitertes Schema und Mutationen	127
7.1	Schema-Modularisierung	127
7.1.1	Technische Separation	127
7.1.2	Domain-Separation	131
7.1.3	Resolver Map aufteilen und Models durch Context verteilen	133
7.2	Mutationen	136
7.2.1	Ein Produkt erstellen	136
7.2.2	Produkt löschen	139
7.2.3	Wunschliste mit Input-Typen erstellen	141
7.2.4	Wunschlisten kaskadierend löschen	145

8	Implementierung mit Java I: Das erste Schema	149
8.1	Use Case	150
8.2	Das Projekt aufsetzen	150
8.2.1	Den GraphQL-Server aufsetzen	151
8.3	Das initiale Schema aufsetzen	154
8.3.1	Objekte im Schema auflösen	154
8.3.2	Feld-Resolver	156
8.3.3	Das Node-Muster	157
8.3.4	Ergebnisse filtern durch Parameter	159
8.3.5	Objekt-Relationen	160
8.3.6	ID-Referenz-basierte Objekt-Relationen	162
9	Implementierung mit Java II: Erweitertes Schema und Mutationen	167
9.1	Selbstdefinierte Skalar- und Geschäftslogik-Felder	167
9.1.1	Skalar-Typ in Schema und POJO definieren	168
9.1.2	Die GraphQLScalarType-Implementierung	169
9.1.3	Geschäftslogik-Felder und -Parameter	172
9.2	Mutationen erstellen und Schemamodularisierung	175
9.2.1	Kunden registrieren	175
9.2.2	Adressen löschen	178
9.2.3	Adressen erstellen: Input-Typen	180
9.2.4	Bestellung erstellen: verschachtelte Input-Typen	182
	Literaturverzeichnis	187
	Index	191

1 API-Grundlagen

»APIs sind überall.«

Martin Reddy [46]

Wie der heutige Software-Engineering-Manager bei Apple, Martin Reddy, in seinem Buch *API-Design for C++* bereits passend bemerkte, umgeben uns APIs in der modernen Applikationswelt mehr denn je. Der allgemeine Trend, immer mehr Softwareprodukte als einzelne Komponenten zu verpacken und öffentlich oder intern zur Verfügung zu stellen, wird vor allem deutlich, wenn man online nach API-Verzeichnissen Ausschau hält.

Die Onlineplattform ProgrammableWeb [44] verwaltet so ein Verzeichnis bereits seit 2005 und kann seitdem ein durchschnittliches Wachstum von fast 2.000 neu registrierten APIs pro Jahr verzeichnen [49].

Doch wieso gibt es in den letzten Jahren einen derart hohen Anstieg an registrierten APIs? Um diese Frage beantworten zu können, hilft es, sich mit den Grundlagen des Konzepts Application Programming Interface vertraut zu machen.

In diesem Kapitel wird die Frage beantwortet, was ein API eigentlich ist und welche Vorteile es bietet. Neben einer klaren Definition des Begriffs, wird auch ein Einblick in drei technische Umsetzungen Teil dieses Kapitels sein. Das Kapitel ist für Entwickler bestimmt, die sich vorher noch nicht oder nur wenig mit dem Thema API beschäftigt haben oder Grundlagen auffrischen möchten.

1.1 Was ist ein API?

API steht für *Application Programming Interface* – oder auf Deutsch: *Anwendungs-Programmier-Schnittstelle*. Die Idee des API ist schon relativ alt: 1952 formulierte David Wheeler, Informatikprofessor an der Universität Cambridge, einen Leitfaden zum Extrahieren einer Subroutinen-Bibliothek mit einheitlichem, dokumentiertem Zugriff [59]. Mit dieser Arbeit legte er den Grundstein für die Idee heutiger APIs.

Der Begriff selbst kam zum ersten Mal in einem Konferenzbeitrag von Ira Cotton und Frank Grestorex auf der *Fall Joint Computer Conference* im Jahr 1968 auf [13]. Darin erweiterten sie die Idee von David Wheeler insofern, als dass die Implementierung und Schnittstelle der Subroutinen-Bibliothek konzeptionell voneinander zu trennen sind. Somit kann die logische Komponente ohne Einfluss auf den Benutzer der Schnittstelle ausgetauscht werden.

»APIs sind wie User Interfaces – nur mit anderen Nutzern im Fokus.«

David Berlind [8]

APIs sind wie Nutzer-Schnittstellen, nur für andere Nutzer konzipiert, so Berlind. Während ein User Interface (UI) die Interaktion eines Menschen mit dem System ermöglicht, soll ein API anderen Systemen diese Interaktionen gewähren. Es ist also anders als eine UI kein »*human-readable interface*«, sondern ein »*machine-readable interface*«.

In einem abstrakten Beispiel kann man sich ein API als Steckdose vorstellen. Computer, Staubsauger, Mixer, Toaster – ganz unterschiedliche Geräte nutzen den Service »Strom« des Serviceanbieters »Steckdose« für verschiedene Zwecke. Dadurch werden diese Geräte Konsumenten eines Service über einen einheitlichen Serviceanbieter.

Damit dieser Zugriff der unterschiedlichen Systeme auf dieselbe Steckdose funktioniert, müssen bestimmte und bekannte Muster, wie die Geräte verbunden werden können, sowie genaue Spezifikationen des angebotenen Service vorliegen.

Die genaue Spezifikation des Service hinter der Schnittstelle Steckdose ist dabei, wie in Abbildung 1–1 zu sehen, bestimmt von Daten, wie etwa 230 V Spannung auf 16 A Stromstärke. Das stellt die Art des Service dar, legt also fest, was genau über das API versendet wird, und definiert seine Repräsentation – also in welcher Form der Service im System des Servicenutzers integriert werden kann.

Die Form der Steckdose legt dabei fest, wie genau ein Nutzer den Service ansprechen muss, um die entsprechende Repräsentation des Stroms integrieren zu können. Die Steckdose ist daher in diesem Beispiel die eigentliche Schnittstelle – also das API. Die Form ist seine technische Umsetzung. Wie im Beispiel von Abbildung 1–1 können dabei mehrere gleiche Steckdosenformen eine Abstraktion für unterschiedliche Servicespezifikationen sein. Repräsentation und technische Umsetzung sind also unabhängig voneinander.

Wie bei der Steckdose bietet auch das API einen Service, der von unterschiedlichen Konsumenten in genau spezifizierter Form – also Repräsentation – über eine genau definierte Schnittstelle angefordert und aus-











							
Spannung	230 V	230 V	400 V	400 V	230 V	400 V	500 V
Strom	16 A	16 A	16 A	32 A	16 A	16 A/32 A/63 A	125 A
Phase	1ph AC	1ph AC	3ph AC	3ph AC	1ph AC	3ph AC	DC
Leistung	2 kW (3,7 kW)	3,7 kW	11 kW	22 kW	3,7 kW	22 kW /44 kW	50 kW
Ladedauer	8 h	4 - 6 h	2 - 3 h	1 - 2 h	4 - 5 h	1 - 2 h	20 min
Verwendung	keine Kommunikation				Kommunikation zu PKW		
	😊	😊	😐	😐	😐	😐	😐
	😊	😊	😐	😐	😐	😐	😐
	😐	😊	😊	😊	😊	😊	😊

Abb. 1-1

Unterschiedliche Steckdosenspezifikationen für die Elektromobilität [52]

geliefert werden kann. In der Anwendung sind das beispielsweise geografische Kartendaten bei Google Maps als Service, die in einer Webseite oder mobilen Applikation für die Anzeige eigener Karten konsumiert werden.

1.2 Vorteile eines API

Doch wieso hat sich die Steckdose in unseren Haushalten so etabliert? Wieso ist das API eine immer wiederkehrende Erscheinung, die sich langsam aber stetig als ein Standard in unserer Applikationswelt etabliert? Wieso sollte man als Stromanbieter seinen Nutzern Steckdosen zur Verfügung stellen?

1.2.1 Flexibilität für Anbieter und Konsument

Eine Steckdose ist bequem. Mit ihr kann sich ein Hersteller von Toastern auf die Entwicklung des Toasters konzentrieren, ohne sich dabei Gedanken über die Herkunft des Stroms machen zu müssen. Solange er die standardisierte Schnittstelle der Steckdose benutzt, wird dieser ihm vom Anbieter auf vereinbarte Weise geliefert.

Für den Anbieter des Stromes heißt das auch, dass er sich ebenso keine Gedanken über die Integration seines Services in die Systeme der Nutzer machen muss. Er stellt ihn lediglich in bekannter Form, direkt an der Schnittstelle zur Verfügung und kann dahinter wesentliche Teile des Stromnetzes – wie die Quelle des Stroms oder die Farbe der Stromkabel – verändern, ohne dabei die Konsumenten funktionsuntüchtig zu machen. Zwischen Stromanbieter und -abnehmer findet also eine Entkopplung statt.

1.2.2 Einheitliches Design und Funktionen

Durch die große Beliebtheit mobiler Endgeräte bieten die meisten Unternehmen ihre Daten und Funktionen nicht nur auf ihrer Webseite an. Eine Kundenbindung über unterschiedliche Kanäle ist ein gängiges Geschäftsmodell für moderne Unternehmen.

Den eigenen Service über ein API zur Verfügung zu stellen, kann dabei helfen, diesen auf diverse Kanäle zu transportieren. Die lose Kopplung der Schnittstellen ermöglicht, die gleichen Daten und Funktionen an verschiedene Abnehmer zu verteilen und dadurch sowohl auf mobilen Endgeräten, der Webseite, bis hin zu Smarthome-Geräten den Endnutzern ein einheitliches Design und dieselbe Funktionsweise zu bieten.

1.2.3 Neue Geschäftsfelder

Vor einigen Jahren machte Amazon seine ersten Umsätze durch den Onlineversand von Büchern. Mittlerweile gilt das Unternehmen als einer der größten IT-Giganten unserer Zeit. Möglich wurde das nicht durch die Aufnahme zusätzlicher Waren in den Shopkatalog, sondern durch ein Umdenken in der Firmenkultur. Als Produkt werden mittlerweile nicht nur die Waren im Onlineshop gesehen, sondern Services, die Amazon zusätzlich bieten kann.

Diese unter dem Namen Amazon Web Services (AWS) zusammengefassten Dienste bieten von Webhosting über Cloud-Services eine breit gefächerte Auswahl an Produkten über eine Vielzahl von APIs an, die Amazon bereits für interne Zwecke benötigt. Diese über eine Schnittstelle nach außen zu tragen, verbreiterte also die Geschäftsfelder des Unternehmens signifikant. Unter anderem konnte durch das Anbieten dieser Services mit Netflix ein weiteres milliardenstarkes Unternehmen als Kunde gewonnen werden.

Ähnlich kann der Entwickler eines E-Mail-Diensts diesen entweder individuell in das System jedes einzelnen Kunden integrieren, oder den Service über ein API allgemein entkoppelt zur Verfügung stellen. Damit können Firmen und Bereiche, die eventuell zuvor nicht als Kunden in Betracht gekommen wären, diesen Service selbstständig verwenden.

1.2.4 Innovationstreiber API

Zudem sind APIs Innovationstreiber. Sicher hätte bei Google niemand damit gerechnet, dass mit der Veröffentlichung ihrer Google-Maps-API eine Augmented-Reality-Anwendung Einzug in viele private Smartphones hält. Das Spielprinzip hinter dem von Niantic entwickelten Mobile Game Ingress [35] ist spätestens seit dem Hype um Pokemon Go im Jahr 2016 [36] bekannt: mit dem Smartphone auf einer echten Karte

der Umgebung und mit einer Augmented-Reality-Funktion der Kamera bestimmte Spielziele erfüllen. Öffentliche APIs ermutigen Entwickler, mit dem angebotenen Service ganz neue Wege zu beschreiten. Es findet auch hier eine Erweiterung der Geschäftsfelder und somit eventuell der Einnahmequellen des Anbieters statt.

1.3 API: Die Definition

Was genau veröffentlicht ein API-Anbieter? Was integriert ein API-Konsument in seine eigenen Systeme? Was sind eigentlich API-Anbieter, API-Konsumenten, und was unterscheidet sie? Um diese Fragen zu beantworten, bedarf es klarer Definitionen von APIs und deren Akteuren.

1.3.1 API-Vertrag

Wie bereits erwähnt, sind APIs wie Steckdosen: eine Abstraktion eines bestimmten Service, mit dem sich verschiedene Konsumenten in einer genau festgelegten Art verbinden können. Außer in einer Smarthome-Umgebung sind die Konsumenten eines API jedoch selten Toaster, sondern andere Programme, die die Funktionen des API nutzen oder integrieren möchten.

Martin Reddy verfasste 2011 eine kurze, allgemeine Definition von APIs [46]. Laut Reddy bieten APIs eine Abstraktion für ein Problem und spezifizieren, wie Benutzer mithilfe von Softwarekomponenten, die eine Lösung des Problems implementieren, interagieren sollen. Reddy spricht also davon, dass ein API aus mindestens zwei Teilen besteht: zum einen einer Abstraktion, die hilft, ein genaues Problem zu lösen, zum anderen die Spezifikation, wie mit ihr zu interagieren ist. 2014 ging Joshua Bloch auf Basis dieser Definition noch etwas tiefer ins Detail:

»Ein API spezifiziert die Operationen sowie Ein- und Ausgaben einer Softwarekomponente. Ihr Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren, sodass diese Implementierung variieren kann, ohne die Benutzer der Softwarekomponente zu beeinträchtigen.«

Joshua Bloch [9]

Bloch sieht somit ebenfalls sowohl eine Abstraktion von Funktionen als auch die genaue Spezifikation der Interaktion mit ihr als Teile des API, legt jedoch noch besonderen Wert auf die Trennung von Implementierung und der eigentlichen Schnittstelle.

Daniel Jacobson, Greg Brail und Dan Woods ziehen ähnliche Schlüsse und erarbeiten in ihrem Werk *APIs – A Strategy Guide* [37] folgende, ausführliche Definition für APIs:

API-Definition von Daniel Jacobson, Greg Brail und Dan Woods

Ein API ist ein Weg für zwei Computer, miteinander über ein Netzwerk (überwiegend das Internet) mit einer gemeinsamen Sprache, die beide verstehen, zu kommunizieren. APIs haben bestimmte Eigenschaften:

- Der API-Anbieter beschreibt exakt, welche Funktionalität das API anbietet.
- Der API-Anbieter beschreibt, wann die Funktionalität verfügbar ist und wann sie möglicherweise inkompatibel verändert wird.
- Der API-Anbieter kann zusätzliche technische Beschränkungen des API skizzieren, beispielsweise Zugriffsraten und -grenzen, die kontrollieren, wie oft eine spezifische Anwendung, oder ein Endnutzer es innerhalb einer Stunde, eines Tags oder eines Monats benutzen darf.
- Der API-Anbieter kann zusätzliche rechtliche oder geschäftliche Beschränkungen zur Benutzung des API skizzieren, beispielsweise markenschutzrechtliche Einschränkungen oder Arten der Benutzung.
- Entwickler akzeptieren, das API so zu nutzen, wie es vom API-Anbieter beschrieben ist, und benutzen nur die Teile, die er beschrieben hat, nach den Regeln, die von ihm für die Nutzung festgelegt wurden.

Diese Auflistung an Rechten und Pflichten von API-Anbietern und konsumierenden Entwicklern sichert die Funktionsfähigkeit der Schnittstelle und wird deshalb auch seiner Form entsprechend *API-Vertrag* genannt. Dabei handelt es sich nicht etwa um ein festgelegtes Schriftstück, sondern um einen Begriff für die Erwartungshaltung des Entwicklers sowie das Versprechen des API-Anbieters über das dokumentierte Verhalten des API.

Bricht der API-Anbieter ein Verhaltensmuster des API, indem er zum Beispiel eine bestimmte Objektrepräsentation ändert, bricht er damit den API-Vertrag und stört meist auch die Funktionalität der gegen das API entwickelten Applikationen.

1.3.2 Die Akteure eines API

Mit dem API-Anbieter und konsumierenden Entwicklern wurden bereits zwei Akteure bei der Verwendung eines API genannt. Es gibt in diesem Kontext also unterschiedliche Rollen.

Der *API-Anbieter* ist zumeist – aber nicht immer – auch der *Besitzer des Service*. Der Service wäre im bekannten Steckdosen-Beispiel

der Strom und sein Besitzer daher der Stromanbieter. API-Anbieter wäre derjenige, der die Steckdose bereitstellt – also der Hauseigentümer. Mit dem Google-Maps-API wäre bei einem anderen Beispiel Google sowohl der Besitzer des Service – also dem Kartendienst – als auch der API-Anbieter.

Der *Entwickler* ist sozusagen der Gegenpart zum API-Anbieter, also der API-Konsument. Dieser entwickelt Applikationen mithilfe der Daten und Funktionen des API. Er verwendet direkt die Schnittstelle, ist somit der primäre Konsument.

Die Applikationen des Entwicklers werden letztendlich einem *Endnutzer* zur Verfügung gestellt, der Interesse an den Daten und Funktionen des API hat, jedoch nur indirekt über die Applikationen mit diesen Daten und Funktionen interagiert. Endnutzer sind also die sekundären Konsumenten und meist die eigentliche Motivation für die Entwicklung einer Schnittstelle.

Mehr Informationen zu den Akteuren im Umfeld einer API finden sich in Kapitel 2.1.

1.3.3 Release-Arten von APIs

Der API-Anbieter hat volle Kontrolle darüber, wer sein API benutzen darf und wer nicht. Laut Daniel Jacobson, Greg Brail und Dan Woods [37] kann man von zwei verschiedenen Arten von APIs ausgehen: *privat* und *öffentlich*.

Öffentliche APIs sind für alle Entwickler nahezu ohne Einschränkungen und vertragliche Vereinbarungen mit dem API-Anbieter über ein offenes Netzwerk frei verfügbar. Hierzu zählen die großen und bekannten APIs, etwa von Twitter [58] oder Facebook [16].

Private APIs stellen laut dem CEO von Runscope, John Sheehan, den größten Teil der API-Welt [11]. Öffentliche APIs seien dabei höchstens die Spitze des Eisbergs, während Unternehmen heute sicher achtmal mehr private, als öffentliche APIs entwickeln. Das Konzept des privaten API muss dabei jedoch noch unterschieden werden zwischen *internen* und *Partner-APIs*.

Interne APIs befinden sich meist in abgeschlossenen Netzwerken und sind lediglich für die unternehmenseigenen Entwickler verfügbar. Partner-APIs dahingegen lassen auch zuvor vertraglich geregelte Zugriffe von (externen) Partnerentwicklern zu. Durch die geringere Zahl von Konsumenten können private oder nur für Partner veröffentlichte APIs besser an deren Wünsche und Bedürfnisse angepasst werden.

Die meisten Unternehmen beginnen mit der Entwicklung eines privaten API und veröffentlichen dann nach und nach Teile dessen – entweder erst für Partner oder gleich komplett öffentlich. Das funktioniert

recht einfach, da sich öffentliche und private APIs inhaltlich sowie meist auch technisch nicht unterscheiden. Es liegen jedoch einige Unterschiede bei der Bereitstellung des APIs vor. Private Schnittstellen müssen sicherer verschlüsselt und deren Nutzung authentifiziert werden. Öffentliche APIs hingegen benötigen eine bessere Dokumentation und mehr Vorsicht bei der Wartung. Sollte man die Veröffentlichungsart des API verändern wollen, muss man beachten, dass sich damit auch die Prioritäten ändern, worauf man im Umfeld der Bereitstellung des API besonders Wert legen sollte.

Weitere Informationen zur privaten und öffentlichen API finden sich in Kapitel 2.2.1 sowie 2.2.2.

1.4 Mögliche API-Technologien und -Spezifikationen

Nun stellt sich die Frage, wie genau eine Umsetzung oder Implementierung von APIs aussieht.

Es gibt mittlerweile eine Vielzahl von Paradigmen, Spezifikationen, Frameworks und Technologien, um ein API zu gestalten und zu implementieren. Um eine grobe Einordnung der Vielfalt zu bekommen, lohnt sich ein Blick zurück auf die Geschichte von APIs.

1.4.1 Geschichte der Remote Execution

Schon seit der Entdeckung des Prinzips, ein Netzwerk durch die Verbindung zweier oder mehrerer Computer zu erzeugen, gibt es einen großen Bedarf daran, die Kommunikation dieser Maschinen zu optimieren. Die ersten Betriebssysteme Unix, Linux und Windows verfügten bereits über interne Protokolle für die verteilte Kommunikation. Doch die Herausforderung lag darin, ein generelles Framework für Entwickler zur Verfügung zu stellen.

Die 1990er- bis 2000er-Jahre

Als in den 1990ern TCP/IP zum Standard für Netzwerkkommunikation wurde, verschob sich der Fokus auf plattformübergreifende Interaktion. Eine Maschine konnte unabhängig vom Betriebssystem auf einer anderen Maschine eine Aktion initiieren.

Die COBRA-Bibliothek mit Protokollen und Algorithmen, Microsofts Distributed-Component-Object-Model-(DCOM-) Komponente oder die Java Remote Method Invocation (Java RMI) sowie andere

Konzepte und Frameworks konnten als entwicklerfreundliche Abstraktionsschicht über der Kern-Netzwerk-Infrastruktur verwendet werden. Diese ersten Versuche, technologieunabhängige Kommunikation für maximale Reichweite in einem Cross-Plattform-Netzwerk zu entwickeln, war ein wichtiger Schritt in Richtung der heutigen Client/Server-Architektur.

Anfang der 2000er erfolgte dann die erste Evolution des World Wide Web, wie wir es heute kennen. HTTP etablierte sich dabei vom De-facto-Standard zum offiziellen Standard für die Netzwerkkommunikation. Vor allem mit der Kombination von HTTP und XML hatten Entwickler auf einmal ein selbstbeschreibendes, sprachen- und plattformunabhängiges Framework für die Kommunikation zur Verfügung.

Die Standardisierung dieses Frameworks erfolgte unter dem Namen Simple Object Access Protocol, kurz SOAP. Das ist eine Messaging-Protokoll-Spezifikation für den Austausch von strukturierten Informationen über das HTTP in einem Netzwerk aus verbundenen Webservices. Die Informationsstruktur wird dabei durch die Web Service Definition Language (WSDL) definiert, einem spezifischen XML-Format. Nun hatten Entwickler endlich Interoperabilität über alle Plattformen und Runtimes hinweg.

Das Web 2.0

Als nächster Schritt erfolgte die Entwicklung des Web hin zum sogenannten *programmable Web* oder auch Web 2.0. Es schoben sich statt einfacher, statischer Webseiten immer mehr Webanwendungen mit komplizierterer Architektur ins Internet. Die Interaktion des Nutzers mit der eigenen Webseite war der neue Fokus. Um diese Interaktionsmöglichkeiten zu bieten, war mehr Kommunikation zwischen Clients und Servern nötig. Die Schnittstellen, über die kommuniziert wurde, gewannen entsprechend deutlich an Bedeutung.

Im Vergleich mit der durch SOAP sehr restriktiven Netzwerkkommunikation über WSDL-XML und HTTP kam mit dieser größeren Bedeutung von APIs der Wunsch nach einem freieren Standard auf. Schon in der Webentwicklung wurden JavaScript und der Datenbeschreibungsstandard JavaScript Object Notation (kurz JSON) aufgrund der Freiheit und Flexibilität, die diese Technologien boten, immer beliebter. Der Wunsch lag nahe, diese Vorteile auch auf APIs auszudehnen. Als Resultat löste JSON letztendlich XML als Kommunikationsformat über HTTP ab.

Die Kombination von JSON und HTTP führten zu verschiedenen Interpretationen der Dissertation *Architectural Styles and the Design of*

Network-based Software Architectures [21] des Netzwerkarchitektur-Pioniers Roy Fielding aus dem Jahre 2000. Diese werden heute generell als REST APIs zusammengefasst.

SOAP wurde seitdem nur noch für große Unternehmen verwendet, deren Systeme auf diesen Restriktionen basieren. Für moderne Neuentwicklungen wurde hingegen REST bevorzugt.

Heute und in Zukunft

Heutzutage, man könnte sagen »nach der Web-2.0-Revolution«, warten schon wieder neue Herausforderungen. Während im Web 2.0 Client-Server-Kommunikation und Webapplikationen der Hauptfokus waren, ist heute Microservices das Buzzword schlechthin. Aus dem Architekturstil, Applikationen immer verteilt zu entwickeln, folgt natürlich auch ein Anstieg der Kommunikation zwischen den Applikationen oder kleineren, eigenständigen Einzelteilen einer einzigen.

Auch das nächste große Buzzword ist nicht weit entfernt: das Internet of Things. Der Trend, immer mehr Alltagsgegenstände um eine Rechnerfunktionalität zu erweitern und miteinander zu verbinden, wird auch in Zukunft dafür sorgen, dass Schnittstellen zur Kommunikation zwischen verteilten Applikationen an Bedeutung gewinnen.

Da HTTP entwickelt wurde, um die Cross-Plattform-Kommunikation des Internets zu optimieren, muss die Frage gestellt werden, ob es sich hierbei immer noch um den optimalen Weg der Kommunikation innerhalb unserer verteilten Systeme handelt. Die erneut relevante Suche nach der Verbesserung dieser Kommunikation kann wieder zu neuen Innovationen führen.

Auch künftig werden die diversen Interpretationen von REST wohl die am weitesten verbreitete API-Art im Netz sein. Aber große Firmen und kleinere Communitys arbeiten schon an und mit neuen potenziellen Standards. Die Breite an Alternativen der Technologien und Konzepte war jedoch noch nie so groß. Von Googles gRPC, das das Grundkonzept von SOAP wiederbelebt, über die vom Ember.js-Erfinder und Open-Source-Pionier Yehuda Katz entwickelte JSON:API-Spezifikation bis hin zu Facebooks GraphQL – es halten viele neue Namen Einzug in die API-Welt.

Daher folgt auf den nächsten Seiten eine Übersicht einiger dieser Möglichkeiten der technologischen Realisierung von APIs.

1.4.2 RESTful HTTP

REST steht für *Representational State Transfer* [21]. Es handelt sich dabei nicht um eine konkrete Technologie und keinen Standard, sondern

vielmehr um einen Architekturstil oder ein Programmierparadigma, das 2000 von Roy Fielding in seiner Dissertation mit dem Titel *Architectural Styles and the Design of Network-based Software Architectures* konzipiert wurde.

REST ist ein Konzept, in dem Daten als Ressourcen gesehen werden. Diese können über eine eindeutige Adresse – eine URI – identifiziert werden und sind in einem spezifischen Format – nach Fieldings ursprünglicher Vision XML – repräsentiert. Eine Ressource kann sowohl ein einzelnes Objekt als auch eine Sammlung – also Collection – von Objekten sein (siehe Abbildung 1–2).

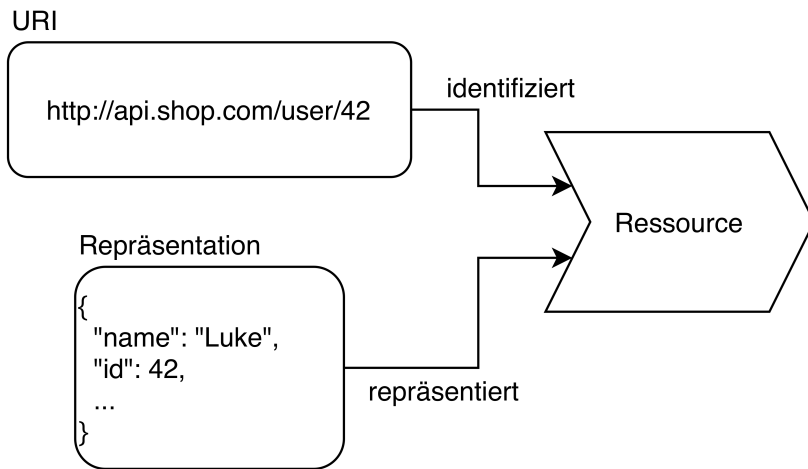


Abb. 1–2
Ressourcen,
Repräsentation und URI
im Zusammenhang

Kommunizieren kann der Client mit den Ressourcen bei einer RESTful API über HTTP und dessen Methoden. Diese sind POST, GET, PUT sowie DELETE und bilden die CRUD-Operationen (Create, Read, Update, Delete). Die Kommunikation findet dabei in einem Request/Response-Muster statt. Der Client stellt über die HTTP-Methoden entsprechende Requests an die gewünschte Ressource des API, und der Server antwortet mit einer Response: bei einem lesenden Zugriff beispielsweise mit der Repräsentation dieser Ressource im Body der Response.

Ein Request auf solch ein RESTful API über die Konsole – mit einem sogenannten cURL-Befehl – würde beispielhaft dann wie folgt aussehen:

```
$ curl -X GET "https://api.shop.com/user/5"
```

Hier fordert der Client einen lesenden Zugriff auf den User mit der ID 5. Auf diesen Request würde er nun vom Server folgende Response erhalten:

Listing 1–1
REST GET Request