

Brandenburgische Technische Universität Cottbus-Senftenberg
Institut für Informatik
Fachgebiet Praktische Informatik/Softwaresystemtechnik

Masterarbeit



Brandenburgische
Technische Universität
Cottbus - Senftenberg

Integrationstesten von GraphQL mittels Prime-Path Überdeckung

Integration testing of GraphQL using Prime-Path Coverage

Tom Lorenz

MatrikelNr.: 3711679

Studiengang: Informatik M.Sc

Datum der Themenausgabe: 16.05.2023

Datum der Abgabe: (hier einfügen)

Betreuer: Prof. Dr. rer. nat. Leen Lambers

Gutachter: M.Sc Lucas Sakizoglou

Eidesstattliche Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Wörtlich und inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Die Arbeit ist nicht in gleicher oder vergleichbarer Form (auch nicht auszugsweise) im Rahmen einer anderen Prüfung bei einer anderen Hochschule vorgelegt oder publiziert worden. Der Verfasser erklärt sich zudem damit einverstanden, dass die Arbeit mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft wird.

.....
Ort, Datum

.....
Unterschrift

Inhaltsverzeichnis

1	Abstract	1
2	Einleitung	3
2.1	Motivation	3
2.2	Umsetzung	4
3	related Work / verwandte Arbeiten	5
3.1	Property Based Testing	5
3.2	heuristisch suchenbasiertes Testen	6
3.3	Deviation Testing	7
3.4	Query Harvesting	7
3.5	Vergleich der Arbeiten	8
3.6	Andere Arbeiten	8
3.6.1	Empirical Study of GraphQL Schemas	9
3.6.2	LinGBM Performance Benchmark to Build GraphQL Servers	9
3.6.3	GraphQL A Systematic Mapping Study	9
4	Grundlagen / Theorie	10
4.1	Graphentheorie	11
4.1.1	allgemeiner Graph	11
4.1.2	Gerichteter Graph	11
4.1.3	Wege und Kreise	12
4.1.4	Erreichbarkeit	12
4.2	GraphQL	13
4.2.1	Schema & Typen	13
4.2.2	vordefinierte Typen	14
4.2.3	Resolver	16
4.3	Zusammenhang Graphentheorie und GraphQL	18
4.4	Testen	23
4.4.1	Arten von Tests	23
4.4.2	Test-Coverage	24
5	Graphcoverage	25
5.1	Graphcoverage allgemein	25
5.2	Graphcoverage Kriterien	26
5.2.1	Edge Coverage	26
5.2.2	Node Coverage	26
5.2.3	Edge-Pair Coverage	26

5.2.4	Prime-Path Coverage	26
5.3	Graphcoverage für Code	26
5.4	Graphcoverage für GraphQL	26
6	Testentwurf	27
6.1	erste Phase / GraphQL Analyse	27
6.1.1	GraphQL in Graph übersetzen	27
6.1.2	Pfadgenerierung	30
6.1.3	Filtern der Prime-Paths	34
6.2	zweite Phase / Pfade untersuchen und Tests für resolver entwickeln	35
7	Testautomatisierung	36
8	Praxis	37
8.1	Toolchain	37
8.2	Ablauf der Generierung	38
8.3	Requirements an das Tool	38
9	zukünftige Arbeit	39
10	Fazit	40
11	Glossar	41
	Literaturverzeichnis	42

1 Abstract

Im Zuge der digitalen Transformation nimmt die Anzahl von Softwareanwendungen rasant zu und insbesondere durch das Internet of Things und die generelle fortschreitende Vernetzung diverser Geräte nimmt in diesem Maße auch die Netzwerklast zu. Bisheriger Standard für Kommunikation von Geräten über das Internet waren REST-APIs diese haben jedoch gewisse Limitierungen wie zum Beispiel: Ineffizienz durch Overfetching/Underfetching, Anzahl an Requests, Versionierung und Komplexität uvm. Mit der Veröffentlichung von GraphQL in 2015 wurde ein Konkurrent zu REST in das Leben gerufen der diese Probleme beheben kann. Durch GraphQL lässt sich insbesondere die Netzwerklast reduzieren da eine GraphQL-Request, im Gegensatz zu REST, mehrere Anfragen in einer einzigen HTTP-Request zusammenfassen kann und dabei auch nur die wirklich gewünschten Daten überträgt. Dadurch, dass jedoch die wachsende Anzahl von Softwareanwendungen auch in immer kritischere Bereiche des Lebens vordringt, ist es enorm wichtig die Qualität der Software sicherzustellen. Eine Methodik zum Sicherstellen der korrekten Funktionalität von Software ist das Testen von Software im Sinne von Validierungstests die sicherstellen sollen, dass die Software vorher definierte Szenarien nach Erwartung behandelt. Für REST-APIs existieren zahlreiche Tools die solche Validierungstests automatisch übernehmen können wohingegen es noch einen Mangel an Tools dieser Art für GraphQL gibt. Im Rahmen der IEEE/ACM 2021 wurde ein Paper veröffentlicht, dass eine Methode vorstellt wie GraphQL-APIs mithilfe von Property-based Tests automatisch getestet werden können. Property-based bezieht sich darauf, dass die Eigenschaften eines Objektes genutzt werden um diese zu testen. Diese Methode generiert zufällig Tests und bietet so eine Möglichkeit, Fehler zu entdecken und generell erstmal eine Grundlage an Tests zu schaffen frei nach dem Motto: lieber irgendwelche Tests als gar keine Tests". Die zufallsbasierte Testgenerierung weist allerdings einige Schwachstellen auf. So kann Sie nicht garantieren, dass die API zu jeder Zeit eine gute Coverage hat denn es können einzelne Routen der API komplett ausgelassen werden. Es sind Testszenarios denkbar, die sehr viele false-positives durchlassen & somit die Qualität der Software nicht ausreichend sicherstellen können. GraphQL ermöglicht außerdem einen potentiell unendlichen Suchraum für die Tests. Um diesem Problem zu entgehen wurde ein einfaches rekursions-limit definiert was jedoch dazu führt, dass die Testabdeckung nur bis zu einem bestimmten Grad überhaupt reichen kann. Längere Pfade die mit GraphQL definiert werden können, werden deshalb niemals getestet.

Mit dieser Arbeit soll ein anderer Ansatz für die automatisierte Testgenerierung untersucht werden. Hierbei soll untersucht werden, inwiefern Graphcoverage auf GraphQL angewendet werden kann zur Generierung von Tests. Mithilfe von einem allgemeinen Graphalgorithmus, der sonst zur Kontrollflussgraphen Analyse benutzt wird, wollen wir zeigen, dass es möglich ist mit einem iterativen Verfahren GraphQL zu testen. Wir

versprechen uns von den Graphcoverage-Algorithmen einer verlässliche und sichere Generierung von Tests die einerseits effizient jegliche Methoden abdecken und andererseits die Anzahl der Tests minimal hält. Um die Methodik zu validieren werden am Ende beide Tools einem Experiment unterzogen um zu sehen ob die hier vorgestellte Methode eine Verbesserung erzielen kann.

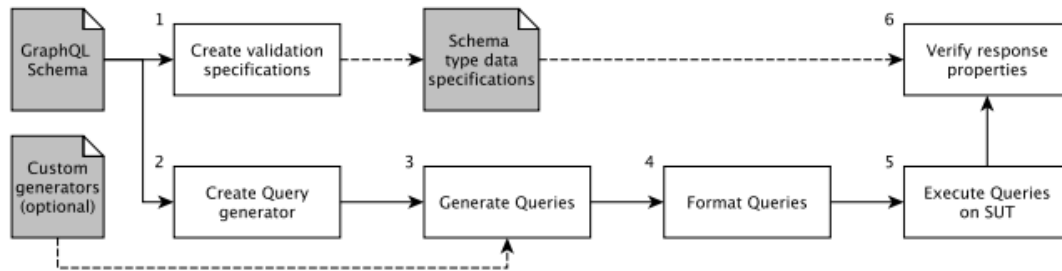
2 Einleitung

In diesem Kapitel wird an das Thema und die Motivation dieser Arbeit herangeführt. Außerdem wird definiert, welche Ziele diese Arbeit erreichen soll und eine grobe Übersicht über die Kapitelstruktur gegeben.

2.1 Motivation

Mit einer steigenden Nutzung von GraphQL wird es immer wichtiger, geeignete Tests für GraphQL-API's zu entwickeln damit eine gute Softwarequalität sichergestellt werden kann. Tests können manuell geschrieben oder automatisch generiert werden. Während bei Unit-Tests für einzelne Methoden der Programmierer frei entscheiden kann, ob er die Tests selbst schreiben will oder doch eher von einem Tool generieren lassen will sind bei Integrations-Tests die Testräume teilweise so groß, dass ein manuelles Schreiben dieser Tests einerseits fehleranfällig aber auch schlicht zu langwierig ist. Für REST-APIs existieren schon automatische Integrationstesttools wie zum Beispiel: EvoMaster [1], QuickREST [2] oder RESTTESTGEN [3]. GraphQL-APIs haben leider noch einen Mangel an solchen automatischen Testtools. Im Rahmen der IEEE/ACM 2021 wurde mit "Automatic Property-based Testing of GraphQL APIs" [4] eine Methode vorgestellt die diesen Mangel angehen soll. Ergebnis der Arbeit war hierbei ein Prototyp der die vorgestellte Methode umsetzen sollte. Die vorgestellte Methode bezieht sich auf "Property-based Testing" wobei diese gleichzusetzen ist mit "Random Testing" [4][vgl. 2.B]. Durch die starke Typisierung und Schema-Definition, welche prinzipiell ein Graph ist, lässt sich in GraphQL sehr einfach auswerten welche Anfragen nun zulässig sind. Die hier vorgestellte Methode nutzt den Fakt, dass zum Beispiel alle möglichen Anfragen immer im Query-Knoten beginnen müssen und somit alle weitergehenden Felder im Schema innerhalb des Query-Knoten definiert sein müssen. Da die Definition des Schemas der eines Graphens entspricht ist jedoch der mögliche Suchraum potentiell unendlich da GraphQL Zyklen innerhalb des Graphens erlaubt. Der unendliche Suchraum wurde durch ein Rekursionslimit begrenzt allerdings wird so eine schlechtere Test-Coverage erreicht.

Die bisher entwickelte Methode funktioniert auf folgende Weise:



Methode von [4]

wobei neben dem vorher erwähnten Rekursionslimit außerdem Punkt 6. "Verify response Properties" kritisch zu betrachten ist da die Auswertung wirklich nur auf die Properties schaut. Dies bedeutet, dass ein zurückgegebens Objekt nur auf seinen Typ überprüft wird aber nicht, ob seine tatsächlichen Rückgabewerte, die exakt erwartet sind. Hierdurch können false-positives entstehen.

Die vorgestellte Methode wollen wir verbessern durch Änderung einiger Ansätze. Hierbei sollen Graphcoverage-Algorithmen zum Einsatz kommen, die mit iterativen Verfahren auch zyklische Graphen ideal überdecken können und dabei immer verlässlich arbeiten, sodass die generierten Tests stets für vergleichbare Ergebnisse sorgen und nicht davon abhängig sind, dass der Zufall eine gute Überdeckung liefert. Zusammenfassend sei gesagt, dass bei der Testgenerierung und Testauswertung Verbesserungen möglich sind und dies Gegenstand dieser Arbeit sein soll.

2.2 Umsetzung

Zuallererst wird in dieser Arbeit etwas Theorie definiert und in Bezug gesetzt. Wir beginnen damit die Graphentheorie als mathematisches Konzept zu definieren denn dieses liegt GraphQL zugrunde. Daraufgehend kommt eine kleine Einführung in GraphQL und dann bilden wir schon die Schnittstelle von GraphQL zur Graphentheorie. Sobald diese Verbindung erfolgt ist können wir uns dem eigentlichen Problem widmen: Wie kann man mithilfe von Graphcoverage-Algorithmen Tests generieren? Hierfür erfolgt ein letzter Theorie-Exkurs über Software-Tests. Mit diesen Grundlagen schaffen wir es dann unsere Methode zu entwickeln und können Sie auch mit "Automatic Property-based Testing of GraphQL APIs" [4] vergleichen. Unsere Methode wird konzeptionell vorgestellt und dann folgen einige Implementierungsdetails sowie ein Vergleich beider Tools durch Experimente. Abschließen wird die Arbeit mit einem Ausblick für zukünftige Arbeit und einem Fazit über unsere erreichten Verbesserungen.

3 related Work / verwandte Arbeiten

Da GraphQL eine stetig wachsende Beliebtheit verzeichnet [5][vgl. Language Features] steigt auch der Bedarf und das Interesse an Testmethoden. Aktuell gibt es für GraphQL noch eine Lücke an produktionsreifen Testtools, insbesondere automatischen Testtools. Eine wachsende Anzahl an research-tools beziehungsweise untersuchten Methoden ist allerdings zu verzeichnen. In diesem Kapitel sollen diese Methoden benannt werden und Verwandheiten, Unterschiede oder thematische Überschnitte von dieser und anderen Arbeiten benannt werden.

3.1 Property Based Testing

In "Automatic Property-based Testing of GraphQL APIs" [4] wird der Ansatz des Property-based Testing verfolgt, um Integrationstests zu erstellen. Property-based Testing ist laut dem Paper heute Synonym mit Random Testing" [4][vgl. 2B] wobei zufällig hierbei meint, dass die Eingabedaten und Routen zufällig generiert werden. Wie Eingangs schon erwähnt, hat die Methode einige Limitierungen. Wir wollen diese hier noch einmal aufgreifen und vertiefen. Der allgemeine Funktionsablauf der Testgenerierung laut Paper ist wie folgt:

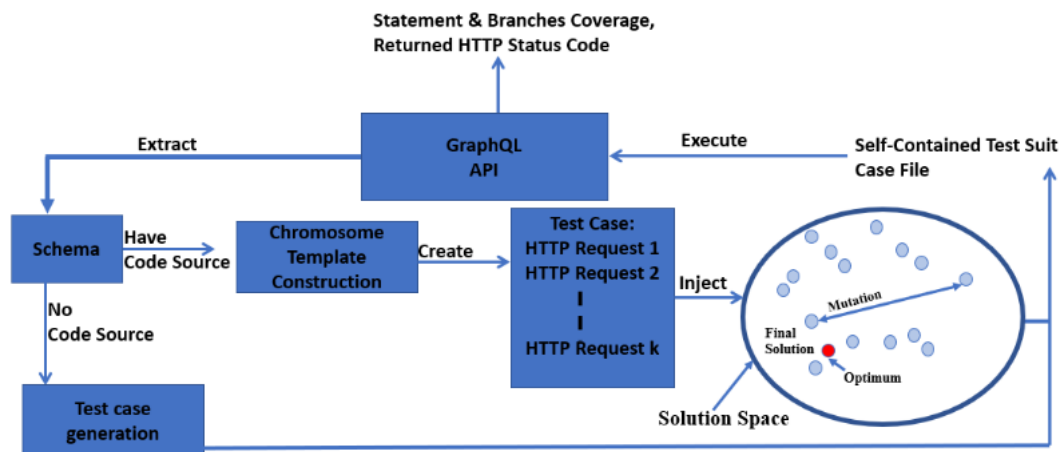
1. Vom Schema, generiere Typ-Spezifikationen
 2. Generiere einen Generator der zufällig eine Liste an Query-Objekten erstellen kann
 3. Generiere n Querys
 4. Transformiere die Queries in GraphQL-Format
 5. Führe die Queries auf dem SUT (system under test) aus
 6. Evaluiere die Ergebnisse auf ihre Properties
- [[4][vgl. 3. Proposed Method]]

Insbesondere Punkt 2. und Punkt 6. weisen Verbesserungsbedarf auf. Punkt 2 wird auch der Hauptunterscheidungspunkt beider Arbeiten sein, denn hier sind dann zwei gänzlich unterschiedliche Konzepte am Werk. Dieser ist beim Property-Based Testing nämlich ein Query-Generator der mithilfe der Clojure-Bibliothek Serene[6] Clojure.Specs[7] generiert und diese Clojure.Specs[7] dann nutzt um mit der Clojure-Bibliothek Malli[8] dann Daten für die Testqueries zu generieren. Unsere herangehensweise wird sich hiervon gänzlich unterscheiden. Im Sinne von Property-based Testing ist diese Herangehensweise allerdings eine sehr sinnvolle gewesen da Malli[8] de-facto Standard für Property-based

Testing in der Clojure-Welt ist. Geht man jedoch davon aus, dass das Ziel eine ideale Überdeckung des Graphens jeder Größe und jeder Struktur ist, so ist diese herangehensweise nicht die beste.[4][vgl. 3C] Laut dem Paper gilt "ein größeres und mehr rekursives (GraphQL)-Schema würde nicht skalieren und der (zufällig) iterative Ansatz ist besser als eine Breitensuche"[4][vgl. 3C]. Diese Behauptung betrachten wir als falsch und behaupten, dass es besser möglich ist. Dies zu zeigen bleibt Gegenstand der folgenden Arbeit.

3.2 heuristisch suchenasbasiertes Testen

EvoMaster[1] ist ein Open-Source Tool welches sich automatisiertes Testen von Rest-APIs und GraphQL APIs zur Aufgabe gemacht hat. Aktuell kann durch EvoMaster sowohl WhiteBox Testing als auch BlackBox Testing durchgeführt werden jedoch ist ein Whitebox Test mittels Vanilla-EvoMaster nur für Rest-APIs möglich die mit der JVM lauffähig sind. Im Paper "White-Box and Black-Box Fuzzing for GraphQL APIs"[9] wurde ein System on-Top für EvoMaster erstellt welches GraphQL Tests generieren kann. Hierbei soll sowohl WhiteBox als auch BlackBox Testing möglich sein. Das erstellte Framework in diesem Paper arbeitet nach folgendem Prinzip:



WhiteBox Testing ist möglich insofern Zugang zum GraphQL-Schema und zum Source Code der API gegeben ist. Andernfalls ist nur BlackBox Testing möglich. Zur Testgenerierung wird ein genetischer Algorithmus genutzt welcher die Tests generiert. Wie dieser genetische Algorithmus genau funktioniert kann im Paper selbst nachgelesen werden[9]. Im Vergleich mit unserer geplanten Arbeit mittels des Prime-Path-Algorithmus ergeben sich einige Unterschiede, diese sind unter anderem: Nutzung eines evolutionären Algorithmus Many-Independent-Objective (MIO). Im Paper selbst wird davon ausgegangen, dass andere evolutionäre Algorithmen unter Umständen passender wären als der MIO Algorithmus für die Testgenerierung. Jedoch ist ein evolutionärer Algorithmus auch immer ein stochastisch, heuristisch sich dem Optimum annähernder Algorithmus. (Beleg

hierfür) Im Gegensatz dazu ist der Ansatz dieser Arbeit ein iterativer Algorithmus der ideale Überdeckungen auf direkte Art bietet und im ersten Durchlauf direkt sein ideales Ergebnis ermittelt. Die ideale Lösung bezieht sich hierbei auf bestimmte Code-Coverage Kriterien die durch unseren Algorithmus erfüllt werden. Inwiefern der evolutionäre Algorithmus diese Kriterien erfüllt bleibt offen, es ist jedoch davon auszugehen, dass er sich einer idealen Lösung dieser Kriterien nur annähert da er eben ein stochastischer Algorithmus ist. (beleg oder Quelle)

3.3 Deviation Testing

Da GraphQL dynamisch auf Anfragen reagiert und es somit möglich ist, in seiner Anfrage einzelne Felder mit einzubeziehen oder auch auszuschließen ist dies im Grunde genommen ein einzelner Test-Case. Im Paper "Deviation Testing: A Test Case Generation Technique for GraphQL APIs" wird diese Gegebenheit benutzt und aus einer selbstdefinierten Query werden hier einzelne Test-Cases gebildet. Ein solcher Test macht je nach Implementierung der GraphQL-Resolver durchaus Sinn, da im Backend Felder durchaus zusammenhängen können und es Bugs geben kann wenn Resolver fehlerhaft definiert sind. z.B. könnte folgende Definition zu solchen Fehlern führen:

(hier BSP mit Code einfügen)

Da Deviation Testing jedoch nur bestehende Tests erweitert um mögliche Felder mitzutesten werden hier keine neuen Tests generiert. Durch Deviation Testing werden bestehende Tests nur erweitert allerdings muss eine Edge-Coverage gegeben sein damit diese Arbeit ein zufriedenstellendes Ergebnis erzeugt. Eine Edge-Coverage in einem komplexen Graphen ist allerdings sehr wahrscheinlich schwer umsetzbar mit manuellem Test schreiben. Eine Paarung von Edge-Coverage mit Deviation-Testing wäre sicherlich Interessant. Genau so wäre es interessant Deviation Testing als Teil unserer Arbeit zu nutzen indem mit diesem Tool die Tests erweitert werden. (initialer Plan war es, einfach immer alle Felder eines Nodes zu testen, hierdurch wäre es möglich auch alle Varianten noch zu testen)

3.4 Query Harvesting

Klassisches Testen von Anwendungen beinhaltet, dass möglichst das komplette System getestet wird bevor es verwendet wird. Im Paper "Harvesting Production GraphQL Queries to Detect Schema Faults" wird ein gänzlich anderer Ansatz verfolgt. Hierbei ist es nicht wichtig die gesamte GraphQL-API vor der Veröffentlichung zu testen sondern echte Queries die in Production ausgeführt werden zu sammeln. Der Ansatz der hierbei verfolgt wird begründet sich daraus, dass ein Testraum für GraphQL potentiell unendlich sein kann und es sehr wahrscheinlich ist, dass nur ein kleiner Teil der API wirklich intensiv genutzt wird, sodass auch nur dieser Teil wirklich stark durch Tests abgedeckt werden muss. AutoGraphQL läuft hierbei in zwei Phasen wobei in der ersten Phase alle einzigartigen Anfragen geloggt werden. In der zweiten Phase werden dann aus den geloggten Anfragen Tests generiert. Hierbei wird für jede geloggte Query genau ein Test-

Case erstellt. Bei dieser Art des Testens wird insbesondere darauf Wert gelegt, dass es keine Fehler im GraphQL Schema gibt. Dies ist ein wichtiger Teil um GraphQL-API's zu testen allerdings noch kein vollständiger Test denn hier wird außer Acht gelassen, dass eine Query konform zum GraphQL-Schema sein kann aber trotzdem falsch indem zum Beispiel falsche Daten zurückgegeben werden durch falsche Referenzierung oder ähnlichem. In dem zu entwickelndem Tool sollten alle Querys die von AutoGraphQL geloggt werden auch berücksichtigt werden da sie durch den Prime-Path Algorithmus auch ermittelt werden. Es kann allerdings sinnvoll sein AutoGraphQL als Monitoring-Software mitlaufen zu lassen und weitere etwaige Fehler hiermit zu loggen und automatisch daraus Test-Cases erstellen zu können damit zukünftig keine Fehler dieser Art mehr passieren.

3.5 Vergleich der Arbeiten

Arbeit / Kriterium	Property Based Testing	heuristisch suchen-basiertes Testen	Deviation-Testing	Query Harvesting
Generierungsart	Zufallsbasierte Routengenerierung	Heuristische Suche	Erweiterung von bestehenden Tests	Tracken von Querys und daraus Tests generieren
Überdeckung	Zufällig, stark abhängig von Schema	abhängig ob Zugang zu Source Code, Zufällig aber optimaler	stark abhängig von selbst geschriebenen Tests	stark Abhängig von User-requests
Orakel	simples Raten	mit Source Code: Analyse	Aus entwickelten Tests	Aus gestellten Querys
Ausführzeit	vor Prod	vor Prod	vor Prod	Verifikation / Wartung
Use-Case	allgemeines Testen	allgemeines Testen	allgemeines Testen	Testen bei Code-Änderung

3.6 Andere Arbeiten

Hier ist eine kurze Übersicht über andere Arbeiten, dieses Kapitel ist sehr unwahrscheinlich in einer Abgabeverision. Es dient eher als Notizensammlung in einer hübscheren Form.

3.6.1 Empirical Study of GraphQL Schemas

Eine umfangreiche Untersuchung von Praktiken in GraphQL. Unterteilt in verschiedene Metriken wie z.B. Anzahl der Objekttypen, Querys etc. Interessant ist allerdings die Untersuchung von zyklischen Schemas. Insbesondere, wie groß diese Zyklen werden können und wie sie begrenzt werden. Dies ist interessant für spätere Auswertungen. Allerdings bringt diese Arbeit nicht viel für das direkte Testen.

3.6.2 LinGBM Performance Benchmark to Build GraphQL Servers

Eher eine Untersuchung wie GraphQL-APIs unter Last performen bzw. wie Effizient sie sind. Der benutzte Query-Generator kann interessant sein aber es ist schwer einschätzbar wie dieser in unserem Kontext genutzt werden kann.

3.6.3 GraphQL A Systematic Mapping Study

Richtig gute Übersicht wie GraphQL Unter der Haubefunktioniert

4 Grundlagen / Theorie

Das automatisierte Testen von GraphQL-APIs erfordert ein spezifisches Domänenwissen in verschiedenen Teilbereichen der Informatik und Mathematik. Dieses Domänenwissen wird in den folgenden Abschnitten aufgrund Lage zweier Lehrbücher erarbeitet und in Kontext gesetzt. Wissen über die Graphentheorie wird benötigt, da GraphQL eine Implementierung von graphenähnlichen Strukturen ist und wir somit Algorithmen darauf anwenden wollen beziehungsweise können. Die mathematische Formalisierung hilft hierbei dann insbesondere bei der Beweisführung für eine allgemeine Termination der zu entwickelnden Algorithmen. Desweiteren ist es nötig sich bewusst zu machen, welche Arten des Testens von Software es gibt und wann man "genug" oder "zu wenig" getestet hat. Hierfür gibt es Metriken die auch insbesondere auf Graphen angewendet werden können und uns dann verraten können ob unsere Tests ausreichend sind. Im konkreten ist das Theorie-Kapitel so strukturiert, dass erst einmal die mathematischen Grundlagen der Graphentheorie vermittelt werden und im Anschluss dazu wird eine Beziehung zwischen GraphQL & Graphentheorie hergestellt. Mit der Beziehung können wir dann zeigen, dass Graphalgorithmen auch bei GraphQL anwendbar sind. Darauffolgend zeigen wir verschiedene Kriterien die eine Testüberdeckung ermöglichen würden und erklären die Unterschiede.

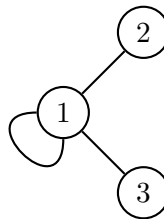
4.1 Graphentheorie

Da GraphQL es ermöglicht, dass komplexe Beziehungen innerhalb eines Datenmodells in Form von Graphen modelliert werden, benötigen wir die Graphentheorie, da diese Methoden liefert um Graphen systematisch zu definieren und analysieren. Desweiteren sind Teile der Testcoverage-Kriterien eng mit Graphenstrukturen verbunden sodass wir gezwungen sind uns einführend mit der Graphentheorie zu beschäftigen damit wir die Grundlagen erarbeiten, die wir später benötigen werden. Generell wird es im folgenden eher etwas theoretischer gehalten, die Zusammenhänge werden sich in späteren Kapiteln erschließen wenn wir Graphen mit GraphQL verbunden oder Tests aus Graphenstrukturen entwickeln.

4.1.1 allgemeiner Graph

Ein Graph ist ein Paar $G = (V, E)$ zweier disjunkter Mengen mit $E \subseteq V^2$ [11, vgl. S.20.1 Graphen]

Elemente von V nennt man Knoten eines Graphens, die Elemente von E nennt man Kanten, Knoten die in einem Tupel von E vorkommen nennt man auch inzident (benachbart) [11, vgl. S.3 0.1 Graphen]. Einen Graphen kann man nun definieren, indem wir zum Beispiel für $V = 1, 2, 3$ wählen und für $E = (1, 1), (1, 2), (1, 3)$. Dargestellt werden können Graphen, indem man die Elemente von V als, zum Beispiel, Kreis zeichnet und dann alle Kanten aus E einzeichnet, indem man die Punkte verbindet [11, vgl. S.2 0.1 Graphen]. Eben definierter Graph hat dann folgende Darstellung:



Es finden sich auch andere Darstellungsformen für Graphen, hier sei insbesondere die Adjazenzmatrix genannt. Für unseren Anwendungsfall belassen wir es jedoch bei Punkten und Linien. Mit dieser Definition lassen sich nun beliebig große, ungerichtete Graphen erstellen.

4.1.2 Grundbegriffe der Graphentheorie

Verschiedene Begriffe sind grundlegend für ein weiterführendes Verständnis der Graphentheorie. Wir werden hier auf einige dieser Grundbegriffe näher eingehen, insbesondere auf diejenigen die im Testkontext unbedingt benötigt werden.

Kante

Die Kante wurde zwar zuvor schon definiert als $e \in E \subseteq V^2$ also ein Tupel (x, y) wobei $x, y \in V$ mit E und V wie in 4.1.1 definiert. Hierbei zeigt die Kante, dass die Knoten

x und y miteinander verbunden sind. Äquivalent ist auch die Aussage, dass y und x verbunden sind. Wir nennen diese Kanten auch ungerichtete Kanten [11, vgl. S.3 0.1 Graphen] Wir wollen nun zwei weitere Arten von Kanten definieren.

gewichtete Kante

Eine gewichtete Kante fügt einen dritten Parameter zu einer Kante hinzu, sein Gewicht. Dies ist ein dritter Parameter einer Kante (x, y) wir erweitern also die Definition um einen dritten Parameter (x, y, z) wobei z meist $z \in \mathbb{R}$ ist. Solche gewichte können benutzt werden um ideale Wege in Graphen zu finden da sie einer Kante eine Zahl zuweisen, die als Kosten genutzt werden können. Gewichtete Kanten werden in unserem Prototypen eine wichtige Rolle spielen allerdings wird bei uns die Kante keine Zahl aus \mathbb{R} als Gewicht bekommen sondern Objekte aus GraphQL.

gerichtete Kante / gerichteter Graph

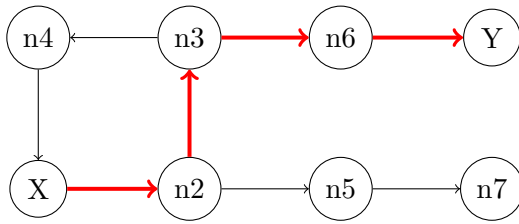
Eine gerichtete Kante besitzt im Gegensatz zu den ungerichteten Kanten eine Orientierung. Dies bedeutet, dass die Kante (x, y) einen Startknoten und einen Endknoten definieren muss. Gekennzeichnet wird dies für den Startknoten durch $init()$ und für den Endknoten durch $end()$. Eine gerichtete Kante vom Knoten x nach y wird somit definiert als $(init(x), end(y))$ [11, vgl. S.26 Verwandte Begriffsbildungen] Die Einführung von gerichteten Kanten ermöglicht es uns, gerichtete Graphen zu erstellen. Gerichtete Graphen sind Graphen, die nur gerichtete Kanten besitzen. Im Prinzip funktionieren gerichtete Kanten wie Einbahnstraßen - es ist nur möglich in die erlaubte Richtung vorzuschreiten. Die gerichteten Graphen sind besonders wichtig für uns, da alle späteren Analysen von GraphQL auf gerichteten Graphen basieren.

Grad eines Knoten

Der Grad eines Knoten gibt an, wie viele andere Knoten mit diesem Knoten verbunden sind und wird als $d_G(node)$ benannt. In ungerichteten Graphen ist der Grad eines Knoten x mit 4 Kanten also $d_G(x) = 4$. Bei gerichteten Graphen unterscheiden wir zwischen Eingangs- und Ausgangsgrad. Der Eingangsgrad ist definiert als Anzahl aller Kanten die auf den Knoten eingehen und der Ausgangsgrad ist genau entgegengesetzt, die Anzahl aller Kanten die ausgehend vom gewählten Knoten sind.

Pfad / Weg

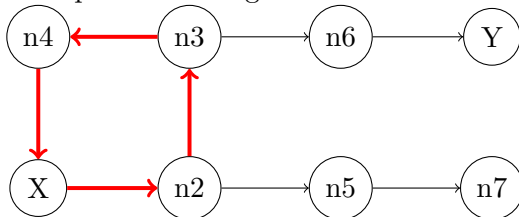
Ein Pfad, oft auch Weg genannt, ist eine Sequenz von Knoten die nacheinander durch Kanten miteinander verbunden sind. [11, vgl. S. 7 0.3] Der Pfad von einem Knoten x zu einem Knoten y über Knoten n_1 bis n_7 hat dann zum Beispiel diese Struktur $p = \{(x, n2), (n2, n3), (n3, n6), (n6, y)\}$ wobei hierbei gilt, dass die Kanten $\{(x, n2), (n2, n3), (n3, n6), (n6, y)\} \in E$. Graphisch würde dies wie folgt aussehen (der Pfad in Rot markiert):



Pfade haben immer eine Länge. Die Länge des Pfades entspricht der Anzahl der Kanten innerhalb dieses Pfades, das bedeutet, dass $\text{Pfadlänge} = |p|$ wobei $p = \{(x, n2), (n2, n3), (n3, n6), (n6, y)\}$. Zu beachten ist, dass Pfade in gerichteten Graphen immer die Orientierung der Kanten beachten müssen. In obigem, gerichteten Pfad wäre also ein Weg $\{(X, N4), (N4, N3)\}$ nicht zulässig.

Zyklus / Kreis

Ein Zyklus, auch Kreis genannt, ist ein Pfad bei dem gilt, dass der Startpunkt und Endpunkt der Pfadsequenz identisch ist. Formell bedeutet dies, dass ein Pfad eine Sequenz $p = \{(X, e1), (e2, e3), \dots, (e_n, X)\}$ ist wobei es auch möglich ist, dass der Kreis Länge 1 hat also die Sequenz $p = \{(X, X)\}$ ist. Zyklen können in gerichteten und ungerichteten Graphen auftreten wobei das Vorkommen in ungerichteten Graphen eigentlich garantiert ist, insofern eine Kante existent ist. Das Auftreten in ungerichteten Graphen ist nicht garantiert aber kann durchaus geschehen. Der Graph aus vorigem Beispiel definiert exemplarisch den Pfad $p = \{(X, n2), (n2, n3), (n3, n4), (n4, X)\}$ welcher ein Kreis ist. Graphisch hervorgehoben:



4.1.3 Zusammenfassung

Wir haben nun eine kleine mathematische Einführung in das Gebiet der Graphentheorie hinter uns. Mithilfe der hier erarbeiteten Begriffe und Definitionen werden wir im folgenden verehrt arbeiten. Insbesondere wenn wir die Coverage-Algorithmen im Bezug des Testens einführen benötigen wir das hier erarbeitete Wissen.

4.2 GraphQL

GraphQL ist eine Open-Source Query-Language (Abfragesprache) und Laufzeitumgebung die von Facebook entwickelt wurde. (vgl. GraphQL-Spec) Die Besonderheiten von GraphQL sind, dass man mit nur einer einzelnen Anfrage mehrere Ressourcen gleichzeitig abfragen kann und die Daten in diesem Schema stark durch einen Typgraphen definiert sind. So lässt sich einerseits die Effizienz stark erhöhen indem weniger Anfragen gestellt werden die zeitgleich eine höhere Informationsdichte haben. Außerdem erleichtert GraphQL die Kommunikation von Schnittstellen indem die gewünschten Felder schon in der Query definiert werden und direkt den erwarteten Datentyp zusichern. Hier liegt auch der große Vorteil im Vergleich zum direkten technologischen Konkurrenten REST API. Bei REST-APIs sind nämlich für verschiedene Ressourcen auch jeweils eine eigene Anfrage nötig und die Typsicherheit ist nicht so stark gegeben wie bei GraphQL-APIs. Diese beiden großen Vorteile sorgen dafür, dass GraphQL an Popularität gewinnt und zunehmend eingesetzt wird. Im Kontext dieser Arbeit ist ein tiefgreifendes, technologisches Verständniss von GraphQL essentiell, deshalb wird hier eine tiefgreifende Erklärung von GraphQL folgen.

4.2.1 Schema & Typen

Grundlage einer jeden GraphQL-API ist ein GraphQL-Schema. Dieses Schema definiert genau wie die Daten der API aufgebaut sind und welche Informationen existieren. Ein GraphQL-Schema ist eine Sammlung von Typen. Typen sind Objekte einer Datenstruktur. Ein Typ definiert alle Informationen über sich, hierbei wird für jede Information ein Feld angelegt. Das Feld kann entweder ein Standarddatentyp wie String, Integer etc. sein oder ein anderer Typ. Falls das Feld ein anderer Typ ist, so entspricht diese Beziehung einer Kante in einem Graphen. Dies bedeutet, dass eine Abfrage dieses Feldes dann ein Objekt des Types zurückliefert und hier auch die gewünschten Felder definiert werden müssen. Ein sehr einfaches Schema wäre zum Beispiel die Beziehung zwischen Büchern und Autoren. Ein Buch hat hierbei einen Titel und einen Author. Ein Author hat einen Namen und ein Geburtsdatum. Ein Schema für dieses Beispiel sähe wie folgt aus:

```
type Buch {  
  title: String  
  author: Author  
}  
type Author {  
  name: String  
  geburtsdatum: Date  
}
```

Es lässt sich also festhalten, dass ein GraphQL-Typ immer als ein Tupel (**Name**, **Felder**) definiert wird wobei die Felder eine Liste an Tupeln (**Feldname**, **Feldtyp**, **Datentyp**) sind. Hierbei gelten folgende Einschränkungen für die Elemente des Tupels:

Feldname ein eindeutiger Feldbezeichner

Feldtyp gibt Einschränkungen vor, z.B. nicht Null (durch !), Listentyp (durch []) etc.

Datentyp der explizite Typ den das Feld hat, kann Standarddatentyp oder anders definierter Type sein

Wenn ein Typ ein Feld enthält, das kein Standarddatentyp ist so entspricht dieses Feld einer Kante in einem Graphen. Dieses Feld muss dann in einer Query näher definiert werden indem angegeben wird, welche Felder nun vom Typ zu dem die Kante führt, ausgegeben werden sollen.

4.2.2 vordefinierte Typen

Jedes GraphQL-Schema definiert initial mehrere Typen die spezielle Aufgaben haben und nicht vom User überschrieben werden können. Im folgenden werden wir auf einige wichtige dieser Typen eingehen.

Scalar-Types

Grundlegende Datentypen (Standarddatentypen) werden durch Scalar Types ausgedrückt. Scalar-Types repräsentieren einzelne Werte wie z.B. einen Integer, einen String, Boolean Werte oder auch Datumstypen. Ein Scalar-Type kann nicht vom User geändert werden und enthält auch keine anderen Typen (im Gegensatz zu Objekttypen die das können). Es ist möglich auch eigene Scalar-Types festzulegen jedoch sind in der offiziellen Spezifikation von GraphQL lediglich folgende Scalar-Types definiert:

Int 32-bit Integer

Float Gleitkommazahl nach IEEE 754

String frei wählbarer Text (leerer String zählt nicht als non-Null!)

Boolean True;False repräsentiert durch internen boolean-type. Ansonsten 0 1

ID einzigartiger Identifier, intern behandelt wie ein String

Scalar Extensions Rohkonstrukt von dem geerbt werden kann für eigen definierte Typen (vgl. GraphQL-Spezifikation 3.5)

Query-Type

Der Query-Type definiert alle erlaubten Anfragen (Leseoperationen) an die GraphQL-API. Hierbei können Anfragen mit und ohne Eingabeparameter angegeben werden. Die definierten Anfragen haben, wie jeder Typ, einen eindeutigen Bezeichner, welcher dann auch in der zustellenden Query benutzt wird. Die Felder der Antwort hängen hierbei vom Feldtypen ab. Ist das Ergebnis der Query-Definition ein Standarddatentyp, so wird

es direkt ausgegeben. Ist es ein anderer definierter Typ, so muss näher bestimmt werden welche Felder erwartet werden. Nutzen wir das Beispiel der Bücher Autoren weiter, könnte man wie folgt einen Query-Type definieren:

```
type Query{
  # returns a List of all Books
  getBooks: [Book]
  # returns one random Book
  getBook: Book
  # returns the Author from a Book Title
  getBookByTitle(String title): Author
}
```

Mutation

Der Mutation-Type ist das Pendant des Query-Typs. Im Mutation-Type werden alle erlaubten Schreiboperationen definiert. Dies beinhaltet das Erstellen, Aktualisieren und Löschen von Daten in der GraphQL-API. Felder im Mutation-Type haben auch immer einen Rückgabewert, Konvention ist es hierbei, die veränderten Objekte zurückzugeben, sodass der Client als Validierung exakt das Objekt bekommt, welches er sich "gewünscht hat". Die Operationen die mittels Mutation hervorgerufen werden, werden linear abgearbeitet. So ist es in GraphQL möglich, die Operationen miteinander zu verknüpfen. Man stelle sich z.B. vor, dass ein User seine Mail ändern will und gleichzeitig noch seinen Namen. Mit REST wären hier zwei Anfragen nötig, GraphQL kann dies mit einer Anfrage erledigen. Kehren wir wieder auf unser bisheriges Beispiel zurück, ein Mutationstyp hierfür könnte wie folgt aussehen:

```
type Mutation{
  # erstelle einen Author, return den erstellten Author
  createAuthor(name: String!, birthdate: Date): Author
  # erstelle ein Buch, return das erstellte Buch
  createBook(title: String!, author: Author): Book
  # ändere den Titel eines Buches, return des Buches mit geänderten Titel
  changeBookTitle(title: String! , newTitle: String!): Book
  # ändere den Geburtstag eines Autors, return des geänderten Autors
  changeBirthdateFromAuthor(author: Author!, newBirthDate: Date!)
}
```

eine einzelne Mutation kann Pflichtfelder und optionale Felder markieren. Pflichtfelder werden mit einem ! markiert. Diese Felder müssen dann als Argument bei jeder Ausführung mitgesendet werden. Optionale Felder benötigen dies nicht.

Subscription

Eine Besonderheit von GraphQL ist der Subscription-Type. Dieser ermöglicht eine Echtzeitkommunikation zwischen Server und Client, indem mithilfe des WebSocket-Protokolls

eine permanente Verbindung zwischen Server und Client hergestellt wird. Diese permanente Verbindung ermöglicht es dem Server direkt Daten an den Client zu senden ohne, dass der Client dafür eine Anfrage schicken muss. Wie auch in allen anderen Typen definiert der Subscription-Type Felder mit einem Namen und Rückgabebetyp. Im Beispiel der Bücher Autoren wäre hierbei nun denkbar, dass eine Subscription für neue Bücher neue Autoren wünschenswert wäre. Hierdurch folgt dann dieser Subscription-Type:

```
type Subscription{
  # neues Buch veröffentlicht
  newBook: Book
  # ein neuer Author erscheint
  newAuthor: Author
}
```

Ein Client könnte sich auf diese Kanäle nun subscriben. Angenommen, wir wollen immer über ein neues Buch informiert werden so muss der Client diese Anfrage stellen:

```
subscription{
  newBook{
    title
    author{
      name
    }
  }
}
```

Hierdurch wird eine permanente Verbindung hergestellt und immer ein Objekt vom Type Book mit den Feldern title & author gesendet, wenn ein neues erschienen ist. Es ist wie auch in anderen Typen möglich, Felder wegzulassen falls diese nicht. Ein spezieller Client muss gewählt werden, der den WebSocket offen lässt, um den die Nachrichten zu empfangen, dies ist jedoch spezifisch abhängig von dem Projekt (Sprache, gewählte GraphQL Server Instanz).

4.2.3 Resolver

Bisher beschäftigen wir uns vorrangig mit der Strukturierung und Typisierung von GraphQL und den Daten die durch das GraphQL Schema dargestellt werden. Ein wichtiger Baustein fehlt aber noch. Woher kommen die Daten? Wie werden Eingabedaten behandelt? Diese Fragen werden durch die Resolver beantwortet. Ein Resolver ist in GraphQL eine Funktion die zuständig für die Datenabfragen und Strukturierung ist. Im Schema haben wir bisher definiert in welcher Art und Weise wir die Daten haben wollen, der Resolver ist nun dafür zuständig, diese Daten im definierten Format zur Verfügung zu stellen. Die Resolver sind nicht, wie alle vorher benannten Teil von GraphQL offen einsehbar, sondern sind Funktionen einer Programmiersprache. Für jedes Feld im Schema, das Daten enthält, muss ein Resolver implementiert werden, dies umfasst insbesondere

die Query, Mutation und Subscription Typen aber auch alle selbstdefinierten Typen. Die konkrete Implementierung der Resolver hängt von verschiedenen Dingen ab, insbesondere jedoch welchen GraphQL-Server man nutzt und welche Programmiersprache verwandt wird. Ein beispielhafter Resolver für die Query eines Buches anhand seines Titels mit ApolloServer in Javascript könnte folgende minimale Syntax haben:

```
const resolvers = {
  Query: {
    book: (parent, args, context, info) => {
      return getBookByID(args.id);
    },
  },
};
```

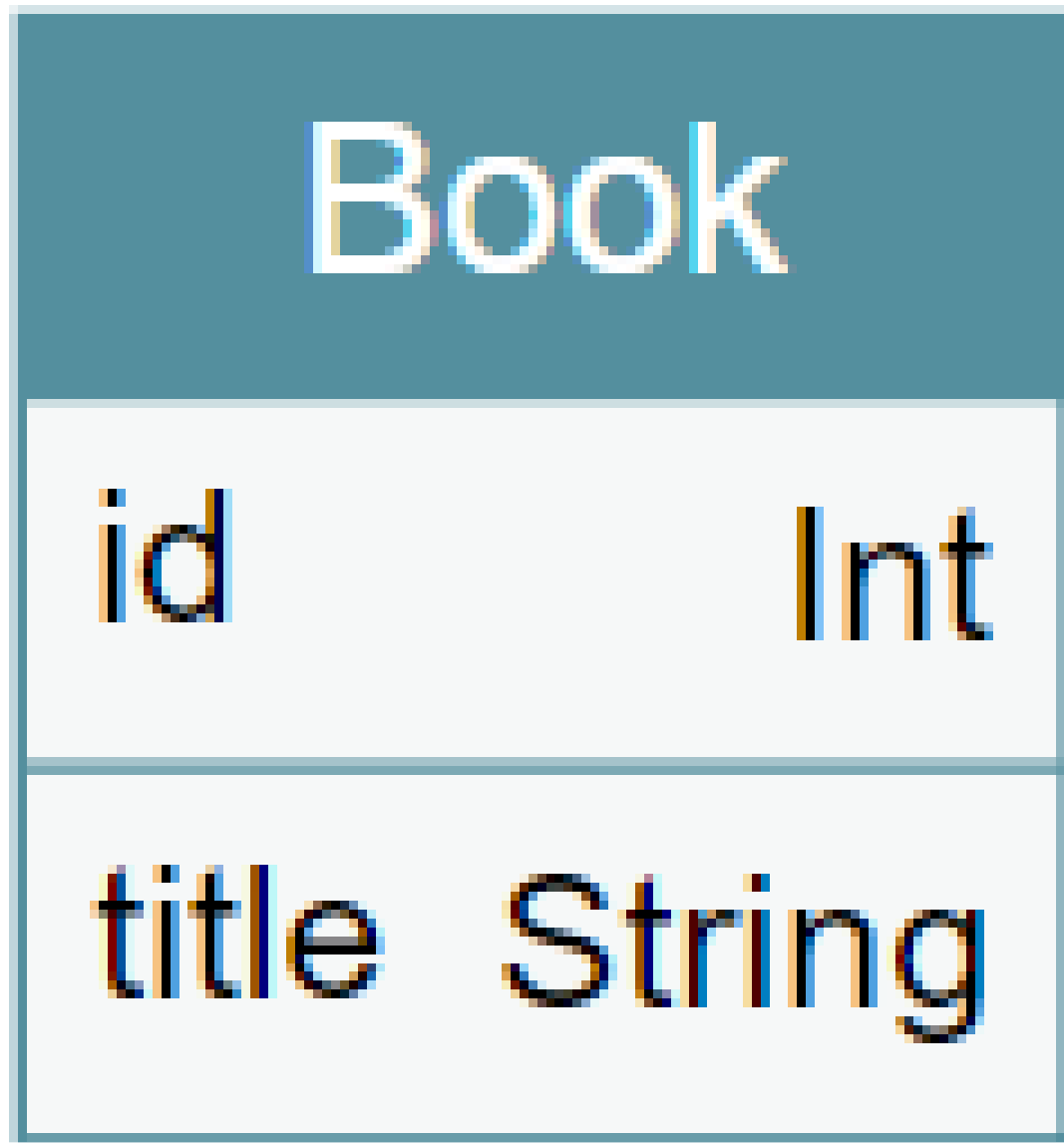
Wobei hierbei zu beachten sei, dass alle Argumente die mitgegeben werden im `args` Argument gespeichert sind. Die Funktion `getBookByID` gibt ein, dem Schema entsprechendes, Json-Objekt zurück. Da die Resolver konkrete Implementierungen außerhalb von GraphQL sind und die einzelnen Resolver untereinander aufrufen können, bedarf es hier einer Reihe an Tests da dieser Code gerne Fehlerhaft sein kann. Ein Klassiker für GraphQL ist es, einzelne Attribute in einem Resolver zu vergessen. Ein Entwickler sollte für jeden Resolver, der ja eine Funktion darstellt, Unit-Tests zur Verfügung stellen. Da sich die einzelnen Resolver aber auch untereinander aufrufen können, muss hier auch die Integrität getestet werden. Hier setzten wir an, indem unsere Testgenerierung darauf aufbaut, jeden möglichen Resolver in mindestens einer Kombination mit anderen Resolv-ern zu testen. So kann sichergestellt werden, dass die Resolver untereinander integer sind.

4.3 Zusammenhang Graphentheorie und GraphQL

GraphQL erlaubt es uns, Typen zu definieren. Ein Type beinhaltet immer mindestens eine Property. Ein Type kann mit einem Knoten eines Graphens gleichgesetzt werden. Und eine Beziehung zwischen Types als Kante. Hierdurch lässt sich dann ein Typgraph entwickeln der als Bauplan für reale Graphen dient. Man nehme zum Beispiel ein Buch und definiere hierfür einen Type

```
type Book {  
  id: Int  
  title: String  
}
```

Es existiert jetzt ein Objekt Book mit den Eigenschaften id als Integer und title als String. Repräsentiert als Graphen hätten wir nun einen einfachen Knoten der zwei Datentypen speichert.



Ein Objekt enthält 1 bis n Property's. Diese Property kann entweder ein Standard-datentyp sein oder auf einen Type verweisen, dies kann der eigene Type oder auch ein anderer Type sein. Fügen wir unserem Beispiel des Buches eine Property hinzu mit dem Type Author wobei der Author selbst wie folgt definiert wird:

```
type Book {  
  id: Int  
  title: String  
  author: Author
```

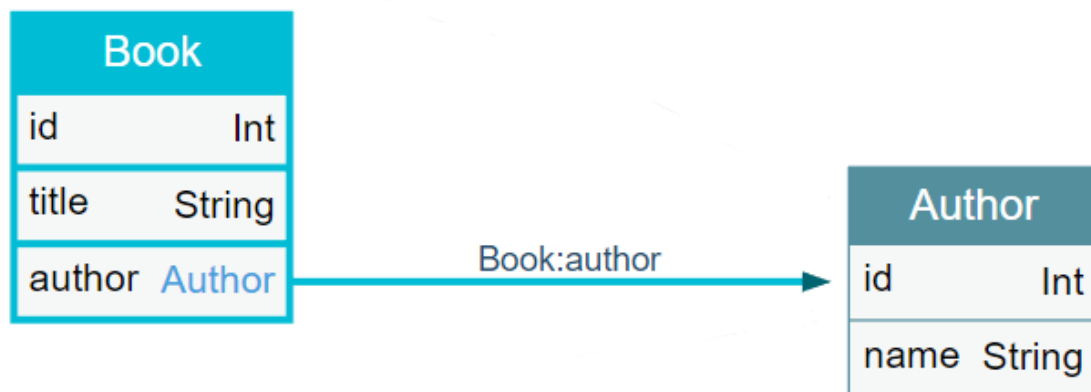


```

}
type Author {
  id: Int
  name: String
}

```

So fügen wir unserem Graphen einen zusätzlichen Knoten hinzu. Die Beziehung zwischen dem Buch und Author wird durch eine gerichtete Kante zwischen dem Buch und dem Author dargestellt. Hierdurch ergibt sich folgender Graph:



Fügen wir dem Author nun auch noch die Property `written` hinzu, so ergibt sich ein Kreis in diesem Graph.

```

type Book {
  id: Int
  title: String
  author: Author
}
type Author {
  id: Int
  name: String
  written: [Books!]
}

```

so ergibt sich, dass wir einen zirkulären Graphen haben mit folgender Struktur

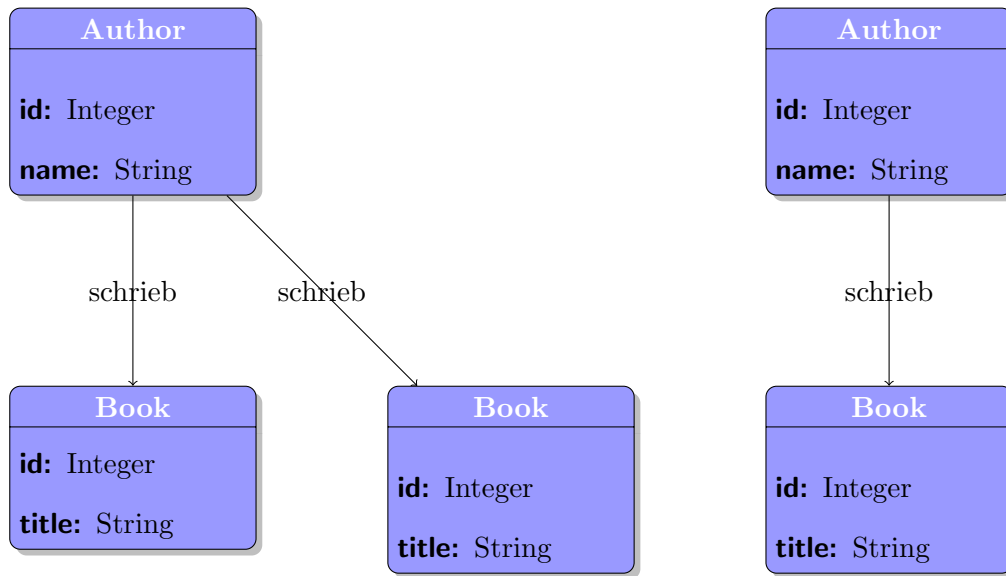


Abbildung 4.1: 3 Bücher, 2 Autoren



Dieses Schema ist ein Bauplan für einen sehr einfachen, zirkulären Graphen der durch GraphQL abgebildet wird. In Realität können die Graphstrukturen die aus diesem Bauplan resultieren sehr unterschiedlich sein. Für dieses Beispiel kann das bedeuten, dass folgende beide Graphen korrekt definierte Graphen sind jedoch komplett andere Möglichkeiten bieten wie man sie abfragen kann. Hierdurch resultiert auch, dass die Abfragen diverse Möglichkeiten haben je nach den unterliegenden Daten.

Definiert man nun, dass es zum Beispiel die StandardQuery "getBookAuthor(id): Author" geben soll. So bedeutet dies, dass ein Author eines Buches abgefragt werden soll aufgrund der Id eines Buches.

(Hier Graphen einfügen mit Highlight der gewählten Knoten)

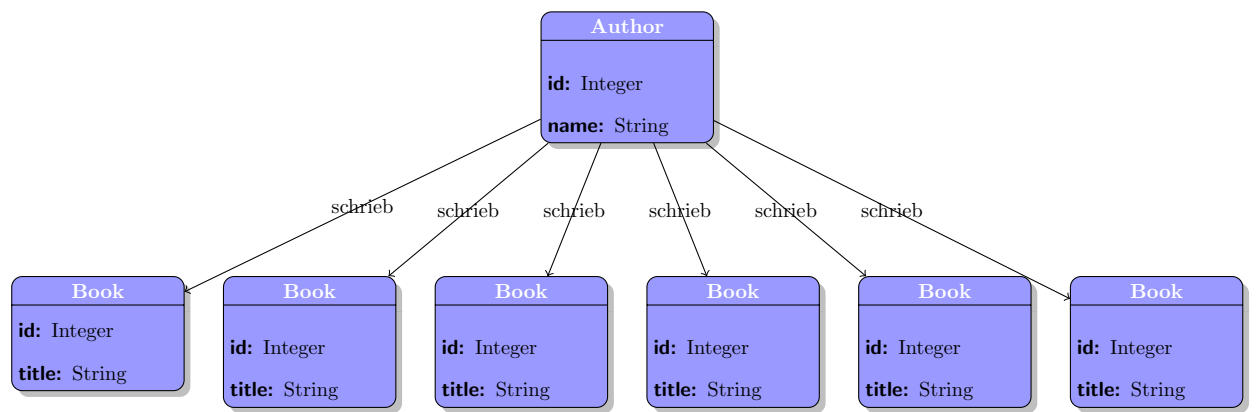


Abbildung 4.2: 6 Bücher, 1 Autor

4.4 Testen

Indem technische Geräte und somit auch Software im umfangreichen Maßstab Einzug nehmen in nahezu alle Bereiche des Lebens ist es wichtig die Sicherheit, Qualität und Zuverlässigkeit von Software sicherzustellen. Um all dies sicherzustellen sind strukturelle Tests von Software nötig. Ziel ist es sicherzustellen, dass die Software den definierten Anforderungen und Spezifikation entspricht. Hierbei sind diverse Techniken und Ansätze verfolgt. Im Rahmen dieser Arbeit wird sich insbesondere auf Integrationstests fokussiert also Tests, die Zusammenarbeit von einzelnen Softwarekomponenten (Units) sicherstellen sollen.

4.4.1 Arten von Tests

Es existieren verschiedene Arten von Tests welche unterschiedliche Ziele haben. Nachfolgend wird eine Auswahl verschiedener Arten erklärt wobei wir in der Granularität aufsteigen, wir fangen mit feingranularen Unit-Tests an und steigen immer weiter zu kompletten System-Tests auf. In der Mitte davon liegt das Integrations-Testen. Diese Methode wird mit dieser Arbeit für einen Anwendungsfall benötigt. Die Erklärung der anderen Arten erfolgt für eine einfachere Einordnung ebendieser.

Unit Testing

Feingranulare Tests die zur Aufgabe haben, einzelne Funktionen/Methoden zu testen. Es wird insbesondere darauf geachtet, dass die einzelne Methode genau das macht, was Sie soll. Es existieren diverse Tools, die diesen Schritt vereinfachen. Diese Tools helfen einem dabei, dass man Tests definieren kann & ein erwartetes Verhalten der Funktion angibt. Eine Auswertung dieser Funktionalität wird dann vom Tool übernommen und bei etwaigen Fehlern gibt es eine Erklärung nach dem Muster: +ABC+ war erwartet, DEF ist eingetreten. Unit-Tests werden bei der Entwicklung der einzelnen Funktionen von der Software entwickelt (sollten sie zumindest). Im Idealfall wird Test-Driven-Development verwendet, hierbei wird erst der Test entwickelt und dann die Methode entwickelt, die diese Tests erfüllen muss.

Integration Testing

Eine Granularitätsebene höher sind die Integrationstests. Hier wird getestet, ob einzelne Komponenten der Software miteinander gut zusammen arbeiten, d.h. ob sie integer sind. Die Testentwicklung ist hierbei meist eher komplex da einzelne Komponenten der Software selbst eine sehr hohe Komplexität haben können. Eine automatische Testentwicklung ist auf dieser Granularitätsebene eigentlich eher selten der Fall, allerdings bietet sich im Kontext von GraphQL die Automatisierung durchaus an da die Zusammenarbeit von Softwarekomponenten mit klar definierter Struktur lohnenswert scheint. Will man allerdings die Intergration von großen Librarys als einzelne Komponente in der eigenen Anwendung testen, so gestaltet sich dies meist schwer und muss manuell getestet wer-

den da meist komplexe Datenstrukturen händisch gemockt (Mocken als Kapitel oder Glossar?) werden müssen.

System Testing

Auf der Ebene des System-Testing wird das komplette entwickelte System getestet. Hierbei ist sicherzustellen, dass alle funktionalen Anforderungen an das System eingehalten werden. Die Tests werden hierbei so realistisch wie möglich ausgeführt, d.h. das Testsystem soll möglichst nah am späteren Produktivsystem sein und Testfälle sollen die funktionalen Anforderungen der Software abdecken. So sind insbesondere alle Geschäftsprozesse zu testen. Im allgemeinen werden System-Tests als Blackbox Tests durchgeführt, dies bedeutet, dass nur externe Funktionsmerkmale getestet werden, z.B. ob eine gewünschte Interaktion stattfand. Hierbei wird keine Rücksicht auf interne Zustände genommen. Ziel ist es, die Funktionsweise so zu testen wie ein realer Benutzer die Software nutzen würde.

4.4.2 Test-Coverage

Es ist nun bekannt, welche Arten des Testens es gibt. Allerdings ist noch nicht klar, wie viele Tests ausreichend sind. Man könnte nun argumentieren, dass man einfach jede einzelne Eingabe in einem Programm testen könnte um zu sehen, dass für jede Eingabe die korrekte Ausgabe kommt. Hierbei bemerkt man jedoch relativ schnell, dass dies mit heutigen Prozessoren nicht mehr möglich ist. Als Beispiel sei hier eine simple Addition von 2 64-bit Integern genannt. Für eine komplette Testung dieser simplen Addition gibt es 2^{64} 18 Trillionen Kombinationen. Mit einem 3GHz Prozessor wäre eine vollständige Testung nach $2^{64} / 3.000.000.000$ 6.149.571 Sekunden (69 Tage) erledigt. Es ist also ersichtlich, dass schon so eine vermeintlich einfache Funktion nicht komplett testbar ist. Daher wird also ein strukturierter Ansatz benötigt, der den Testraum einerseits klein hält, andererseits trotzdem dafür sorgt, dass Fehler ausgeschlossen werden können. Hierfür wurden formale Coverage-Kriterien entwickelt.(vgl. S. 17 software-testing)

Coveragekriterien

5 Graphcoverage

Wie zuvor gesehen, existieren verschiedene Coverage-Kriterien um Testabdeckung zu prüfen. Graphcoverage ist hierbei eine Herangehensweise um graphenbasierte Datenstrukturen zu überdecken. Graphen können nämlich ähnliche Probleme aufweisen wie vorheriges Beispiel der Addition. Die Addition zweier 64-bit Integer ist wenigstens endlich, Graphenstrukturen haben sogar unter Umständen unendliche Testräume. Umso wichtiger ist es hier, dass Überdeckungskriterien formuliert werden können, die diesen möglicherweise unendlichen Suchraum stark verkleinern und dennoch eine ausreichende Testung ermöglichen. Da wir vorher ergründet haben, dass GraphQL sich als gerichteten Graph darstellen lässt, können wir nun die Graphcoverage nutzen, um Tests mithilfe der Graphcoverage zu erstellen. Wie genau die Coverage erstellt wird und daraus Tests resultieren, werden im folgenden geklärt. Gerichtete Graphen sind die Grundlage für viele Coverage-kriterien, wobei die Grundidee hierbei ist, Sachverhalte als Graphen zu modellieren und dann eine ausreichende Überdeckung zu finden. (vgl. Software-testing S. 27 2.1) In (Graphentheorie Kapitel verlinken) wurden gerichtete Graphen bereits erklärt, daher können wir direkt fortfahren und verschiedene Kriterien definieren, die einen Graphen überdecken. Wir erklären zuerst verschiedene Techniken, die einen Graphen überdecken und erklären dann ihre Anwendung an Beispielen. Erst sortieren wir Graphcoverage ein im Kontext von Code-Coverage und bilden im zweiten Schritt eine Coverage für GraphQL.

5.1 Graphcoverage allgemein

Um Graphcoverage zu nutzen, verfeinern wir zuerst die allgemeine Definition von gerichteten Graphen. Ein gerichteter Graph ist ein Paar $G = (V, E)$ zweier disjunkter Mengen (vgl. Graphlehrbuch) mit $E \subseteq V^2$ (vgl. Graphlehrbuch). Die Definition muss hierbeierweitert werden mit :

Menge N von Knoten

Menge N_0 von Anfangsknoten, wobei $N_0 \subseteq N$

Menge N_f von Endknoten, wobei $N_f \subseteq N$

Menge E von Kanten, wobei $E \subseteq N \times N$. Hierbei ist die Menge als $initx \times targety$ definiert.

(vgl. Introduction to Softwaretesting 1-1 Kopie)

Mithilfe dieser Definition können nun z.B. Kontrollflussgraphen abgebildet werden. Eine Definition benötigen wir noch, bevor wir uns den Überdeckungskriterien widmen

können. Ein Pfad, mit möglicher Länge Null, der in einem Knoten N_0 *startet und in einem Knoten N_f endet, ne* 1*Kopie*)

5.2 Graphcoverage Kriterien

Es gibt verschiedene Möglichkeiten die Überdeckung von Graphen zu definieren, hierbei ist Relevant zu betrachten welches Überdeckungskriterium das sinnvollste für das aktuelle System ist, das zu testen ist.

5.2.1 Edge Coverage

5.2.2 Node Coverage

5.2.3 Edge-Pair Coverage

5.2.4 Prime-Path Coverage

5.3 Graphcoverage für Code

5.4 Graphcoverage für GraphQL

6 Testentwurf

Der allgemeine Testentwurf besteht aus zwei einzelnen Phasen. In der ersten Phase wird das GraphQL Schema analysiert und die Prime-Paths generiert. So haben wir grundlegendes Wissen darüber, welche Querys ausgeführt werden müssen damit jeder Bereich der API abgedeckt ist. In der zweiten Phase ...

6.1 erste Phase / GraphQL Analyse

Grundlage der Analyse einer GraphQL-API ist ihr Schema. Die gesamte erste Phase bezieht sich nur auf das Schema denn aus diesem können wir alle grundlegenden Test-Cases ermitteln die nötig sind um eine Überdeckung der Anfragen zu ermitteln. Es sei hier gesagt, dass nur die Überdeckung der Anfragen nicht bedeutet, dass die API hiermit vollständig getestet wird, hierdurch werden nur alle Anfragen erstellt, sodass jeder Knoten und jede Kante im definierten Schema mindestens einmal Betrachtung findet in einem Test. Die dahinterliegenden Resolver benötigen weitere Abdeckung hierzu jedoch mehr in Phase 2.

6.1.1 GraphQL in Graph übersetzen

Um ein GraphQL Schema in einen Graphen zu übersetzen, bedarf es mehrerer Schritte. Im GraphQL Standard implementiert jeder GraphQL-Client eine `parse()` Funktion. Diese werden wir auch nutzen, da wir hierdurch einen Graphen erhalten der für unsere Berechnungen auf dem Graphen passend ist. Die `parse()` Funktion führt im wesentlichen zwei Schritte aus:

Lexikalische Analyse Schema in Token zerlegen

Syntaktische Analyse Token in passende Graph-Repräsentation übersetzen

(Quelle X.Y.Z)

Endergebnis ist ein Abstract Syntax Tree (AST) (Quelle einfügen). Ein AST sieht je nach GraphQL-Client Plattform unterschiedlich, jedoch sehr ähnlich aus. Folgendes, sehr simples Schema:

```
type Query {  
  user(id: Int): User  
}  
  
type User {
```



```

    id: Int
    name: String
}

```

wird in folgenden AST übersetzt (in Java-/Type-script ist der AST in json Format):

```

1 {
2   "kind": "Document",
3   "definitions": [
4     {
5       "kind": "ObjectTypeDefinition",
6       "name": {
7         "kind": "Name",
8         "value": "Book"
9       },
10      "fields": [
11        {
12          "kind": "FieldDefinition",
13          "name": {
14            "kind": "Name",
15            "value": "id"
16          },
17          "type": {
18            "kind": "NamedType",
19            "name": {
20              "kind": "Name",
21              "value": "Int"
22            }
23          }
24        },
25        {
26          "kind": "FieldDefinition",
27          "name": {
28            "kind": "Name",
29            "value": "title"
30          },
31          "type": {
32            "kind": "NamedType",
33            "name": {
34              "kind": "Name",
35              "value": "String"
36            }
37          }
38        }
39      ]
40    }
41  ]
42 }

```

```

40         "kind": "FieldDefinition",
41         "name": {
42             "kind": "Name",
43             "value": "author"
44         },
45         "type": {
46             "kind": "NamedType",
47             "name": {
48                 "kind": "Name",
49                 "value": "Author"
50             }
51         }
52     }
53 ]
54 },
55 {
56     "kind": "ObjectTypeDefinition",
57     "name": {
58         "kind": "Name",
59         "value": "Author"
60     },
61     "fields": [
62         {
63             "kind": "FieldDefinition",
64             "name": {
65                 "kind": "Name",
66                 "value": "id"
67             },
68             "type": {
69                 "kind": "NamedType",
70                 "name": {
71                     "kind": "Name",
72                     "value": "Int"
73                 }
74             }
75         },
76         {
77             "kind": "FieldDefinition",
78             "name": {
79                 "kind": "Name",
80                 "value": "name"
81             },
82             "type": {
83                 "kind": "NamedType",

```

```

84         "name": {
85             "kind": "Name",
86             "value": "String"
87         }
88     },
89     {
90         "kind": "FieldDefinition",
91         "name": {
92             "kind": "Name",
93             "value": "written"
94         },
95         "type": {
96             "kind": "ListType",
97             "type": {
98                 "kind": "NonNullType",
99                 "type": {
100                     "kind": "NamedType",
101                     "name": {
102                         "kind": "Name",
103                         "value": "Books"
104                     }
105                 }
106             }
107         }
108     }
109 ]
110 ]
111 }
112 ]
113 }

```

Die Ausgabe in diesem AST verrät uns für jede „ObjectTypeDefinition“ das wir hier einen Knoten des Graphens haben und in dem Fields-Eintrag kann man alle möglichen Verbindungen des Knotens finden.

6.1.2 Pfadgenerierung

Da wir im vorherigen Schritt eine geeignete Darstellung gefunden haben, um unseren Graphen zu repräsentieren können wir nun im ersten Schritt alle Pfade ausgehend vom Querytype innerhalb dieses Graphens finden. Um die Pfade zu ermitteln müssen wir lediglich wissen, welche Typen ein einzelner Knoten haben kann. In Kapitel (GraphQL Kapitel verlinken) wurde bereits auf alle möglichen Typen eingegangen. Hierbei sind für die Pfadgenerierung nur diese wichtig:

FieldDefinition

NonNullType

ListType

ObjectTypeDefinition

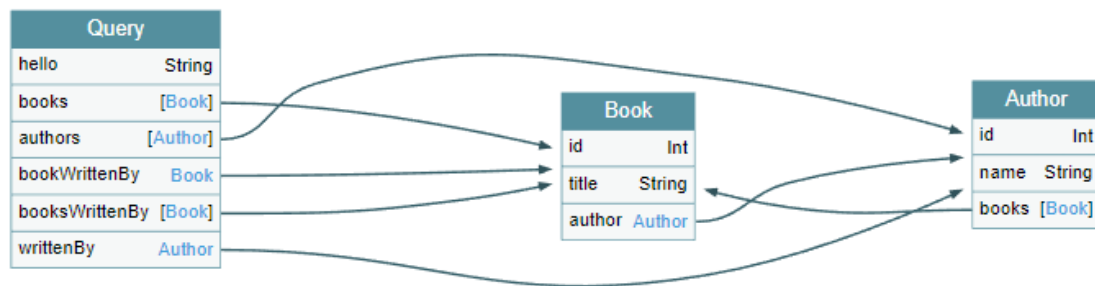
(Quelle X.Y.Z)

Aus unserem schon bekannten Schema:

```
type Book {
  id: Int
  title: String
  author: Author
}
type Author {
  id: Int
  name: String
  books: [Book]
}
type Query {
  # test query
  hello: String
  # all books
  books: [Book]
  # all authors
  authors: [Author]
  # a single book from a author name
  bookWrittenBy(name: String!): Book
  # all books from a author
  booksWrittenBy(name: String!): [Book]
  # a author for a book title
  writtenBy(title: String!): Author
}
```

können wir nun beginnen die Pfade zu generieren. Hierbei müssen wir nur Pfade generieren, die ihren Ursprung im Query Type haben denn andere Typen sind nicht initial abfragbar sondern nur durch Verkettung mit einer Operation vom Query Type. Jedes Feld vom Query Type ist ein Einstiegspunkt für einen Pfad. Ein Pfad führt weiter, wenn das Ergebnis eines Pfades eine ObjectTypeDefinition enthält. Also hier dementsprechend "Author" oder "Book"

Besagtes Schema führt dann zu folgendem Graphen:



Ausgehend davon, dass jedes Feld vom Query-Type Ausgangspunkt eines Pfades ist, existieren in Iteration 0 die Pfade:

Iteration 0:

- hello
- books
- authors
- bookWrittenBy
- booksWrittenBy
- writtenBy

Iteration 1:

- hello
- books
- books → author
- authors
- authors → books
- bookWrittenBy → author
- booksWrittenBy → author
- writtenBy → book

Iteration 2:

- hello
- books

- books \rightarrow author
- books \rightarrow author \rightarrow books
- authors
- authors \rightarrow books
- authors \rightarrow books \rightarrow author
- bookWrittenBy
- bookWrittenBy \rightarrow author
- bookWrittenBy \rightarrow author \rightarrow book
- booksWrittenBy
- booksWrittenBy \rightarrow author
- booksWrittenBy \rightarrow author \rightarrow book
- writtenBy
- writtenBy \rightarrow book
- writtenBy \rightarrow book \rightarrow author

In Iteration 2 ist nun zu erkennen, dass sich Kreise bilden. Weitere Iterationen führen dazu, dass sich nur noch Kreise bilden innerhalb der definierten Struktur. D.h. Iteration 3,4 etc würde an neuen Pfaden nur noch Verlängerungen des Schemas "..... book \rightarrow author \rightarrow book \rightarrow author" hervorbringen. Jede Kante im Graphen entsteht dadurch, dass das Feld keine Standarddatentyp Definition hat sondern einen Objekt-Type. Somit ist es möglich, mithilfe von folgendem Pseudo-Code, die Pfade alle zu erzeugen.

0. Importiere funktionen buildSchema(), parse() und printSchema()
1. Lese GraphQL-Schema String
2. Erstelle AST
3. pfade = []
4. Für alle Definitionen im AST mache:
 - 4.1 Wenn Definition.kind == "ObjectTypeDefinition"
 - 4.1.1 ermitteltePfade = ermittel alle Pfade ausgehend von diesem Knoten
 - 4.1.2 pfade[] = ermitteltePfade
 - sonst ist Definition BasisDatentyp \rightarrow Pfadende
5. return pfade

Endergebnis dieser Funktion sollten exakt Iteration 2 entsprechen.

6.1.3 Filtern der Prime-Paths

Da wir nun eine Liste aller möglichen Pfade haben, müssen wir diese nur noch nach PrimePaths filtern. Wie in (Kapitel verlinken) bereits erwähnt, sind PrimePaths die längsten Pfade, die kein Teilpfad eines anderen Pfades sind. Hierzu können wir eine einfache Funktion entwickeln, die alle nicht Prime Paths herausfiltert. Diese Funktion muss hierfür jeden errechneten Pfad einmal überprüfen ob dieser Pfad ein Teilpfad eines anderen Pfades ist. Sollte der Pfad ein Teilpfad sein, so ist dieser kein PrimePath andernfalls handelt es sich um einen PrimePath und dieser wird behalten. Folgender Pseudo-Code übernimmt die Filterung der Pfade:

```
Input: Pfadliste
0. Variable primePaths: []
1. Für alle pfade aus Pfadliste:
  1.1 Für alle andererPfad aus Pfadliste
    1.1.1 Wenn istKeinTeilpfad(pfad, andererPfad)
      primePaths.push(pfad)
2. return primePaths
# ausgehend davon, dass pfade als Liste [1,2,3] gespeichert sind.
# isSubarray sollte vordefiniert sein in diversen Sprachen
Function istKeinTeilpfad(pfad, andererPfad):
  return !isSubarray(pfad, andererPfad)
```

Wendet man die Filterung nun auf unsere Liste aus Iteration 2 an, sollte sich folgende, gefilterte Liste ergeben:

- hello
- books →author
- authors →books →author
- bookWrittenBy →author →book
- booksWrittenBy →author →book
- writtenBy →book →author

Dies sind dann die Prime-Paths des Schemas. Mithilfe dieser Pfade haben wir jeden Knoten und jede Kante mindestens einmal in einer Query berücksichtigt. Aus dieser Liste können wir nun fortfahren und unsere Tests entwickeln.

6.2 zweite Phase / Pfade untersuchen und Tests für resolver entwickeln

Aus den gefundenen Pfaden entwickeln wir nun die Tests. Wir werden weiterhin das Beispiel von vorher benutzen um zu zeigen, wie die Testentwicklung für unser Schema aussieht. Jeder gefundene Prime-Path wird nun so untersucht, dass er

7 Testautomatisierung

Hi

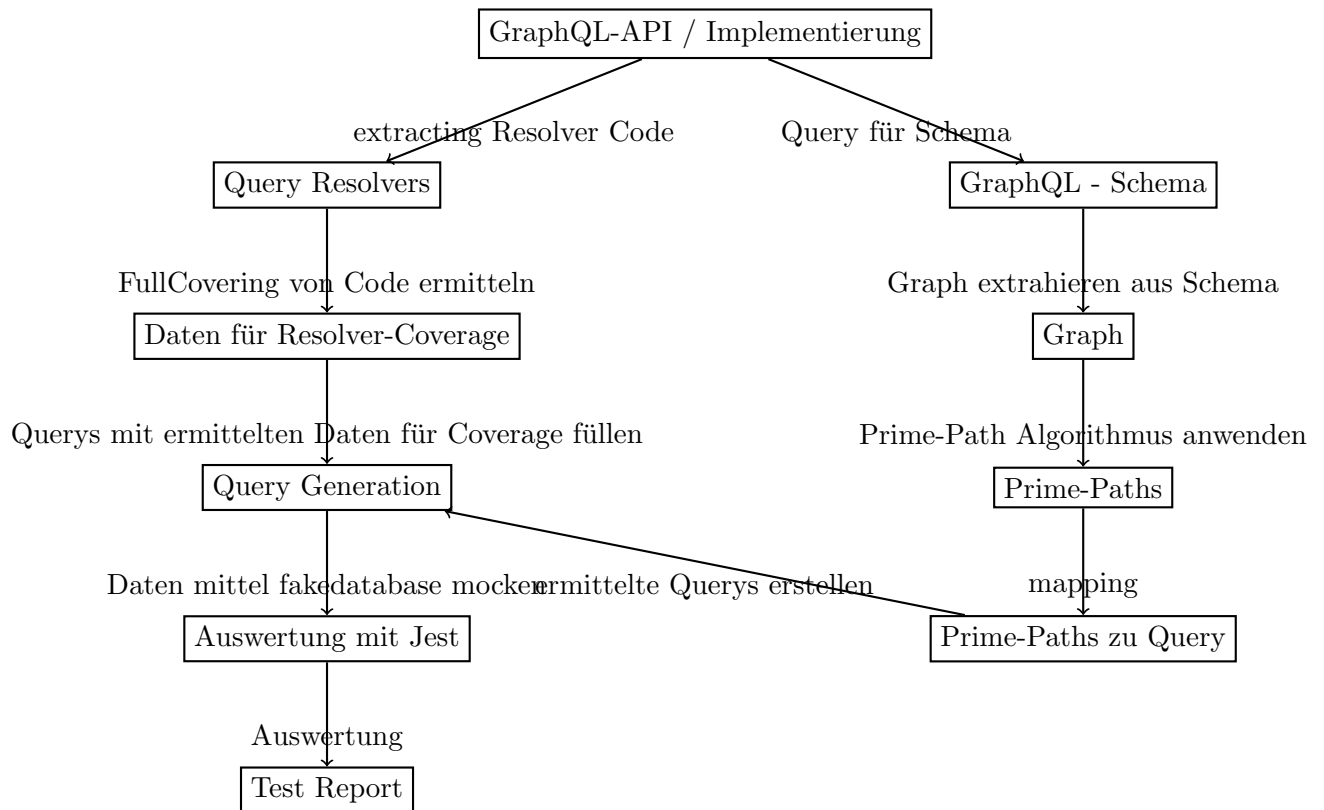
8 Praxis

Im folgenden wird die praktische Umsetzung der vorher erarbeiteten Theorie behandelt. Hierzu wurde/wird ein Tool geschrieben, das aufgrund eines GraphQL-Schemas automatisiert Tests erzeugt. Diese Tests werden dann ausgeführt und ausgewertet mit den entsprechenden Verbesserungen.

8.1 Toolchain

Da GraphQL ein Standard für diverse Sprachen ist und das mocken von Daten essentiell zum testen ist, kann der Teil der Testgenerierung und Auswertung nur sprachspezifisch stattfinden. Es können somit nicht alle Sprachen berücksichtigt werden. Da GraphQL vor allem in der Webentwicklung verwendet wird, bezieht sich das Testtool auf JavaScript/TypeScript mit der Testbibliothek Jest. Als Server für die Verarbeitung wird ApolloServer genutzt, es ist jedoch denkbar, dass man jeden Server einsetzen kann insofern dieser eine `executeOperation()` implementiert die einen String als Query akzeptiert. Um Daten zu generieren wird auf das Tool `Factory.ts` zurückgegriffen (kann sich noch ändern), dieses ermöglicht es Baupläne anzulegen und dann beliebig viele Objekte zu erschaffen. Die benötigte Toolchain ist also sehr klein, sie benötigt nur `Factory.ts` für die Datengenerierung, Jest für die Ausführung der generierten Tests und ApolloServer für die Ausführung der GraphQL-Resolver.

8.2 Ablauf der Generierung



8.3 Requirements an das Tool

9 zukünftige Arbeit

Hisdfsdf

10 Fazit

Hi

11 Glossar

Im Text werden einige Fachbegriffe genutzt. Hier findet sich deren Erklärung

Begriff Erklärung

IEEE/ACM

HTTP

HTTP-Request

API

REST

GraphQL Waren-Management-System; Ein System das das Lager verwaltet und die kompletten Betriebsprozesse eines Lagers abbilden kann

Overfetch

Underfetch

Evolutionärer Algorithmus

Literaturverzeichnis

- [1] *Evo Master*, <https://github.com/EMResearch/EvoMaster>, zuletzt besucht: 15.06.2023.
- [2] S. Karlsson, A. Causevic und D. Sundmark, *QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs*, 2019. arXiv: 1912.09686 [cs.SE].
- [3] *Rest Test Gen*, <https://github.com/SeUniVr/RestTestGen/>, zuletzt besucht: 15.06.2023.
- [4] D. S. Stefan Karlsson Adnan Causevic, “Automatic Property-based Testing of GraphQL APIs,” *International Conference on Automation of Software Test*, 2021.
- [5] *State of GraphQL 2022 survey*, <https://blog.graphqleditor.com/state-of-graphql-2022>, zuletzt besucht: 15.06.2023.
- [6] *Serene - Clojure.Spec from GraphQL Schema*, <https://github.com/paren-com/serene>, zuletzt besucht: 15.06.2023.
- [7] *Clojure Spec - Data structure definition*, <https://clojure.org/guides/spec>, zuletzt besucht: 15.06.2023.
- [8] *Mali - Data Driven Specification Library for Clojure*, <https://github.com/metosin/malli>, zuletzt besucht: 15.06.2023.
- [9] A. Belhadi, M. Zhang und A. Arcuri, *White-Box and Black-Box Fuzzing for GraphQL APIs*, 2022. arXiv: 2209.05833 [cs.SE].
- [10] P. A. J. Offutt, *Introduction to Software Testing*. 2008, ISBN: 978-0-521-88038-1.
- [11] R. Diestel, *Graphentheorie*. 2000, ISBN: 3-540-67656-2.

Onlinere Ressourcen wurden am 16. Juni 2023 auf ihre Verfügbarkeit hin überprüft.