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

Masterarbeit

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 Sakizloglou

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 **2**
2.1 Motivation . . . 2
2.1.1 Testgenerierung . . . 3
2.1.2 Testauswertung . . . 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 . . . 8
3.6.2 LinGBM Performance Benchmark to Build GraphQL Servers . . . 8
3.6.3 GraphQL A Systematic Mapping Study . . . 8

4 Grundlagen / Theorie **9**
4.1 Graphentheorie . . . 10
4.1.1 allgemeiner Graph . . . 10
4.1.2 Gerichteter Graph . . . 10
4.1.3 Wege und Kreise . . . 11
4.1.4 Erreichbarkeit . . . 11
4.2 GraphQL . . . 12
4.2.1 Schema & Typen . . . 12
4.2.2 vordefinierte Typen . . . 13
4.2.3 Resolver . . . 15
4.3 Zusammenhang Graphentheorie und GraphQL . . . 17
4.4 Testen . . . 21
4.4.1 Arten von Tests . . . 21
4.4.2 Test-Coverage . . . 22

5 Graphcoverage **23**
5.1 Graphcoverage allgemein . . . 23
5.2 Graphcoverage Kriterien . . . 24
5.2.1 Edge Coverage . . . 24

5.2.2 Node Coverage 24
5.2.3 Edge-Pair Coverage 24
5.2.4 Prime-Path Coverage 24
5.3 Graphcoverage für Code 24
5.4 Graphcoverage für GraphQL 24

6 Testentwurf **25**
6.1 erste Phase / GraphQL Analyse 25
6.1.1 GraphQL in Graph übersetzen 25
6.1.2 Pfadgenerierung 28
6.1.3 Filtern der Prime-Paths 32
6.2 zweite Phase / Pfade untersuchen und Tests für resolver entwickeln 33

7 Testautomatisierung **34**

8 Praxis **35**
8.1 Toolchain 35
8.2 Ablauf der Generierung 36
8.3 Requirements an das Tool 36

9 zukünftige Arbeit **37**

10 Fazit **38**

11 Glossar **39**

Literaturverzeichnis **40**

1 Abstract

Mit zunehmender Popularität von GraphQL werden Tools benötigt, die Sicherheit, Zuverlässigkeit und korrekte Funktionalität von GraphQL verifizieren und validieren können. Durch die starke Typisierung von GraphQL lässt sich der Testraum jedoch sehr schön eingrenzen. Aktuell gibt es aber noch Einschränkungen an Testtools für GraphQL-Apis, insbesondere im Bereich der automatischen Testanalyse. Im Paper Äutomatic Property-based Testing of GraphQL APIs ̈(Quelle hinzufügen) wurde sich mit einem automatischen Testverfahren für Integrationstest für GraphQL-API's beschäftigt. Hierbei wurde der Ansatz der zufallsbasierten Testgenerierung genutzt, um Tests zu generieren. Die zufallsbasierte Testgenerierung weist allerdings einige Schwachstellen auf. So kann Sie nicht garantieren, dass die API zu jeder Zeit eine gute Coverage hat. 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 wieder nicht garantiert sichergestellt werden kann. 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. Hierbei werden die Probleme des unendlichen Suchraumes und der zufälligen Testüberdeckung systematisch und effizient gelöst. Um zu validieren, dass unsere Methode eine Verbesserung bietet werden einige Experimente durchgeführt in denen wir verschiedene Metriken vergleich hierbei unter anderem: Anzahl der generierten Tests, Testabdeckung der Schnittstelle und gefundene Fehler.

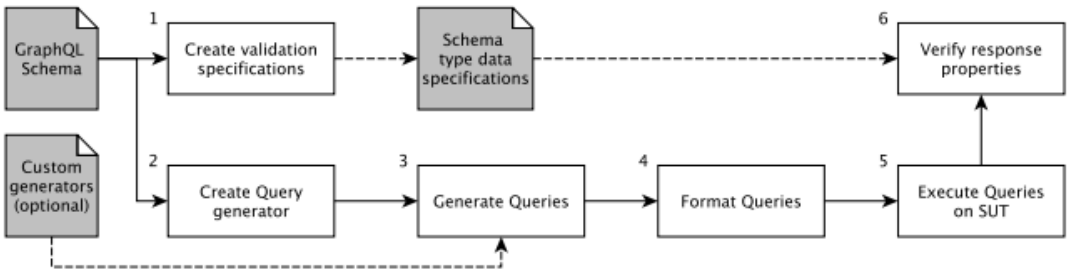
2 Einleitung

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. Idealerweise können diese Testtools solche API's automatisch testen, so wie es für REST-API's schon umgesetzt wurde. Die Struktur von GraphQL erlaubt allerdings zyklische Strukturen und ermöglicht somit ein potentenziell unendlich großen Testraum. In dem Paper Äutomatic Property-based Testing of GraphQL-API's "(hier Quelle) wurde versucht ein solches automatisches Testtool zu entwickeln. Ergebniss der Arbeit war hierbei ein Prototyp der in der Lage ist eine GraphQL-Schnittstelle zu testen allerdings mit zwei technischen Einschränkungen. Die erste technische Limitierung liegt in der Lösung des potentiell unendlichen Testraumes, hierbei wird ein Rekursionslimit festgelegt, dass die maximale Pfadlänge festlegt und somit für einen endlichen Suchraum sorgt. Diese Arbeit soll zeigen, dass die erste technische Limitierung lösbar ist durch einen spezifischen Algorithmus. Eine zweite Limitierung ist die Auswertung der Tests. GraphQL liefert eine stark typisierte Antwort, die vorhersehbar durch die Schemadefinition ist. Im Testtool wird allerdings nur die Typsierung getestet. Dies bedeutet, dass eine gewisse Anzahl an false-positives existieren können geschuldet daraus, dass nur der Typ eines Objektes getestet wird, jedoch nicht seine Exakten Attribute.

Der allgemeine Ablauf des bestehenden Tools ist wie folgt:



Verbesserungen in dieser Arbeit sind insbesondere in den Punkten 2 und 6 (6 wenn genug Zeit) geplant.

Create Query Generator (Punkt 2) Kapitel Testgenerierung

Verify response properties (Punkt 6) Kapitel Testauswertung

2.1.1 Testgenerierung

Der bisherige Ansatz der Testgenerierung ist eine zufallsbasierte Suche. Hierbei wird ein GraphQL-Schema geladen und nach dem Query-Type gefragt. Der Query-Type definiert alle erlaubten Anfragen an die API. Das Ergebnis einer jeden Anfrage ist (ein Knoten) / (eine Liste von Knoten). Jeder Knoten kann dann verwandte Knoten haben. Eben diese werden dann mit zufallsbasierter Suche abgefragt, jedoch nur bis zu einer bestimmten Pfadlänge die durch das Rekursionslimit festgelegt ist, eben um unendliche Suchräume zu vermeiden. Nun ist offensichtlich, dass es durchaus auch Pfade geben kann, die länger als das Rekursionslimit sind und somit nicht vom Testtool berücksichtig werden. Im Sinne einer guten Testcoverage wollen wir aber möglichst jede Funktion mit Tests überdecken, somit erreicht die bisherige Methode leider nicht eine zufriedenstellende Lösung im Sinne der Testcoverage. Diese Arbeit soll die bisherige Methode verbessern indem Schritt 2, der Query-Generator, verbessert wird mit einem endlichen Algorithmus der Graphen jeder Größe und Struktur zuverlässig abdeckt. Hierfür müssen verschiedene Überdeckungskriterien erst definiert werden, allerdings sei schon zu sagen, die hier vorgestellte Methode hat als Ziel, das jede Kante und jeder Knoten des Graphens hierdurch mit mindestens einem Test abgedeckt werden, sodass wir eine erhebliche Verbesserung in der Zuverlässigkeit der Testcoverage erlangen.

2.1.2 Testauswertung

Die Auswertung der Tests nach Äutomatic Property-based Testing of GraphQL-API's ërfolgt durch einen Typabgleich von Query und Response. Eine Validierung der Response wird zeitgleich mit dem erstellen der Querys erledigt. Hierdurch folgt die Limitierung, dass das Testtool aus dem GraphQL-Schema wissen kann, welchen Typ eine Antwort hat, allerdings ist nicht erschließbar, welche genauen Attribute eine Rückgabe hat. So kann eine Anfrage, die als Typ `Automarke` hat, jede Automarke akzeptieren. Sähe die Anfrage allerdings so aus: `getMarke("Opel Corsa")` und die Antwort `Marke(name:Audi)` dann wäre hier eigentlich ein Fehler, das Testtool würde aber akzeptieren, da die Typzuordnung zutreffend ist. Es wäre besser, wenn das Testtool nicht nur den Typ der Response auswertet sondern auch ihren Inhalt. Ob dies umgesetzt wird in dieser Arbeit wird sich zeigen (Zeitliche Komponente; TODO)

2.2 Umsetzung

Zuallererst wird in dieser Arbeit etwas Theorie definiert und in Bezug gesetzt. Angefangen mit einer allgemeinen Definition eines Graphens im mathematischen Sinne und GraphQL als Schnittstelle, wird dann ein Bezug dieser beiden Themen zueinander hergestellt. Im folgenden wird erklärt, was Software-Testing überhaupt ist und inwiefern dieses mit Graphen zusammenhängt. Hierfür werden insbesondere Graphüberdeckungen betrachtet. Abschließend für den Theorie-Teil folgt eine Erklärung wie Graphüberdeckungskriterien und Algorithmen die diese Kriterien erfüllen können, helfen können um GraphQL-APIs zu testen. Um diese theoretischen Erkenntnisse zu validieren und die eingängliche Behauptung zu beweisen wird dann eine Implementierung des PrimePath Algorithmus erstellt die dann den Query-Generator vom Paper Äutomatic Property-based Testing of GraphQL-API's ërsetzen soll. Hierbei ist das Ziel, große Teile des bestehenden Codes zu nutzen und den Query-Generator nathlos in das Programm einzubinden, sodass gezeigt werden kann, dass unsere hier erarbeitete Methode funktioniert. Um zu beweisen, dass unsere Methode funktioniert folgt ein Vergleich beider Generierungsalgorithmen. Verschiedene Metriken sind hierbei interessant, insbesondere jedoch die Anzahl der generierten Tests, die Dauer der Berechnung aller Tests und die Coverage des Graphens. Insbesondere bei der Anzahl an generierten Test / Coverage erhoffen wir uns, dass durch weniger Tests eine höhere Coverage erreicht werden kann. Außerdem erwarten wir, dass die Berechnungszeit für eine vollständige Coverage geringer sein sollte als bei der zufallsbasierten Suche. Hierfür werden die Experimente an 3 verschiedenen GraphQL-API's ausgetestet, 2 stammen hiervon aus dem originalen Paper und eins wird eigens für unsere Methode entwickelt um an diesem besonders die Limitierungen des ursprünglichen Tools zu zeigen und die Lösung der Limitierungen zu validieren.

3 related Work / verwandte Arbeiten

Da GraphQL eine stetig wachsende Beliebheit verzeichnet (Quelle) steigt auch der Bedarf an Testmethoden. Aktuell gibt es für GraphQL noch eine Lücke an produktionsreifen Testtools, insbesondere automatische Testtools. Eine wachsende Anzahl an research-tools bzw. untersuchten Methoden ist allerdings zu verzeichnen. In diesem Kapitel sollen etwaiige Verwandheiten, Unterschiede oder thematische Überschnitte von dieser und anderen Arbeiten benannt werden.

3.1 Property Based Testing

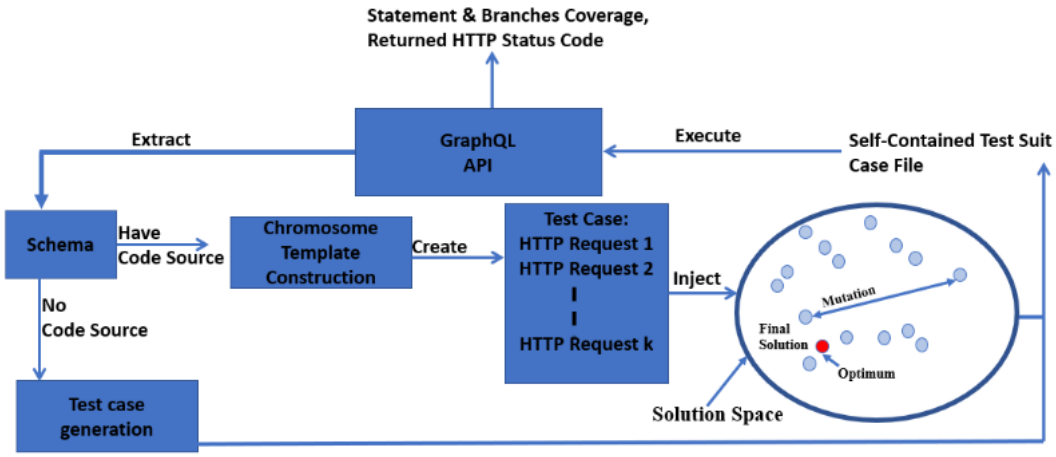
In (Paper Quelle hier verlinken) wird der Ansatz des Property-based Testing verfolgt um Integrationstests zu erstellen. Property-based Testing ist laut dem Paper heute Synonym mit Random Testing"(vgl. Paper) wobei zufällig hierbei meint, dass die Eingabedaten Routen zufällig generiert werden. Der allgemeine Ablauf 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
   (vgl. Paper Seite 3.)

Wie Eingangs erwähnt soll Ziel dieser Arbeit sein, Punkt 2 und vielleicht Punkt 6 zu verändern. In der Entwicklung des Tools wurde Bezug darauf genommen, dass die Random-Testing Methode durchaus ihre Limitierungen hat (maximale Pfadlänge = Rekursionslimit) allerdings wurde die Lösung dieses Problems nur kurz untersucht. Hierbei wird gesagt, dass eine Breitensuche innerhalb des Schemas möglich wäre aber nicht skaliert und der iterative Ansatz die Tests schneller und kontrollierter erstellt (vgl Paper Absatz B). Wir wollen zeigen, dass diese Schlussfolgerung falsch ist und wir einen skalierbaren Ansatz entwickeln können der die angesprochenen Probleme lösen kann. Auf die konkrete Implementierung der zufallsbasierten Generierung unsere Verbesserungen wird später noch eingegangen.

3.2 heuristisch suchenbasiertes Testen

EvoMaster 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 ẄHite-Box and Black-Box Fuzzing for GraphQL APIs"(Quelle hier) 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 (hier Quelle). 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 deterministischer Algorithmus der beweisbar ideale Lösungen 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 Ëeviation 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 druchaus 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 beeinhaltet, 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 berücksichtig 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**
Überdeckungs-kriterien	Heuristische Suche	Erweiterung von beste-henden Tests	Tracken von Querys und daraus Tests generieren	
Orakel	simples Raten			
Testverarbeitung				
Testgranularität				

3.6 Andere Arbeiten

Hier ist eine kurze Übersicht über andere Arbeiten, dieses Kapitel ist sehr unwahrscheinlich in einer Abgabeversion. 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 Ünter der Haubefunktioniert

4 Grundlagen / Theorie

Das automatisierte Testen von GraphQL-API's erfordert ein spezifisches Domänenwissen in verschiedenen Teilbereichen der Informatik und Mathematik. Dieses Domänenwissen wird in den folgenden Abschnitten aufgrundlage 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 können. Die mathematische Formalisierung hilft hierbei dann insbesondere bei der Beweisführug 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 warum wir bestimmte Methoden hier eher nutzen als andere. 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. Mit diesen Grundlagen können wir dann zeigen, dass verschiedene Überdeckungsalgorithmen zur idealen Testgestaltung genutzt werden können wobei hier natürlich definiert werden muss, was überhaupt eine Testüberdeckung ist und wann diese ïdealïst.

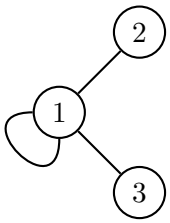
4.1 Graphentheorie

Ein grundlegendes, strukturelles Verständniss von Graphen ist wichtig für diese Arbeit, da diese mathematische Struktur die Grundlage der gesamten Arbeit bildet. In diesem Kapitel werden viele Definitionen mithilfe eines mathematischen Lehrbuchs erarbeitet. Es wird später ersichtlich werden wozu diese Definitionen wichtig sind auch wenn diese jetzt noch sehr abstrakt erscheinen.

4.1.1 allgemeiner Graph

Ein Graph ist ein Paar G = (V, E) zweier disjunkter Mengen (vgl. Graphlehrbuch) mit $E \subseteq V^2 (vgl. Graphlehrbuch)$

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). Ein Graph könnte man nun definieren indem wir z.B. 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 z.B. Kreis zeichnet und dann alle Kanten aus E einzeichnet indem man die Punkte verbindet. Eben definierter Graph hat z.B. folgende Darstellung:



Mit dieser Definition lassen sich nun beliebig große Graphen erstellen. Einige Eigenschaften von Graphen müssen nun noch genau definiert werden da diese später relevant sein werden. Zu definieren sind Wege und Kreise sowie die gerichtete Graphen

4.1.2 Gerichteter Graph

Im Kontext der Überdeckungskriterien sind gerichtete Graphen eher angewandt als ungerichtete, da diese durch ihre Struktur den Programmfluss besser abbilden können. Ein gerichteter Graph G ist definiert als:

FALSCHE DEFINITION: (Ersetzen, normale gerichtete Graphen haben keine Anfangsknoten und Endknoten. Nur die Gerichtetheit bleibt TODO)

Menge N von Knoten

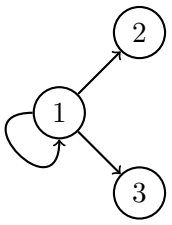
Menge N_0 von Anfangsknoten, wobei $N_0 \subseteq N$

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

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

(vgl. Introduction to Softwaretesting 1-1 Kopie)

Im Grunde ist die Definition sehr ähnlich zu dem eines ungerichteten Graphen. Nur die Defintion der Kanten unterscheidet sich. In einem ungerichteten Graphen hat die Kante (1,2) einen Weg von Knoten 1 zu Knoten 2 und umgedreht. Wohingegen dieselbe Kante in einem gerichteten Graphen nur den Weg von Knoten 1 zu Knoten 2 hat. Für das Beispiel von ungerichteten Graphen also V = 1, 2, 3 und E = (1, 1), (1, 2), (1, 3) ergibt sich also folgender Graph:



4.1.3 Wege und Kreise

Wege

Ein Weg ist eine Sequenz [1,2,...,X] von Knoten, welche alle paarweise adjazent sind. (vgl. Software Testing Intro)

Kreis

Ein Kreis ist ein Weg bei dem Start und Endknoten identisch sind.

4.1.4 Erreichbarkeit

Ein Knoten ist erreichbar von einem anderen Knoten wenn ein Weg von einem zum anderen Knoten existiert.

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 geünschten Felder schon in der Query definiert werden und direkt den erwarteten Datentyp zusichern. Hier liegt auch der große Vorteil im Vergleich zum direkten technologischen Konkurenten 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 exisitieren. 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 bedeuet, 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 Authoren. 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 Einschänkungen 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 intial 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 nich 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-Sepzifikation 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 müss 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-Types. Im Mutation-Type werden alle erlaubten Schreiboperationen definiert. Dies beeinhaltet 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
    createBok(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 Authors
    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 permantente 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 Rückgabetyp. 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);
        },
    },
};
```

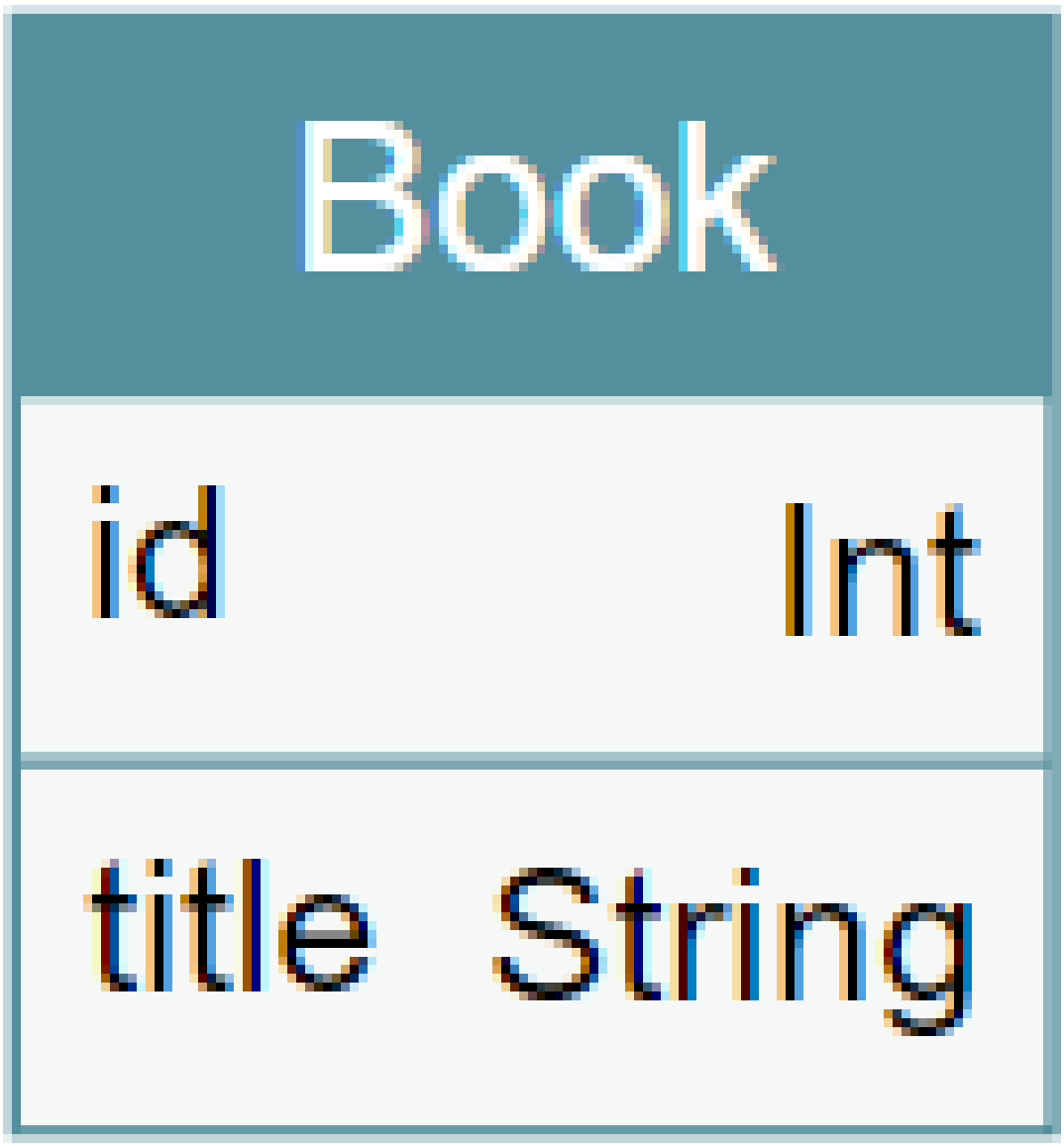
Wobei hierbei zu beachten sei, dass alle Argumente die mitgegeben werden im `args` Argument gespeichert sind. Die Funktion getBookByID gibt ein, dem Schema entsprechendes, Json-Objekt zurück. Da die Resolver konkrete Implementierungen außerhalb von GraphQL sind und die einzelnen Resolver untereinander 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 Resolvern 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 exisitiert 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 Propertys. Diese Property kann entweder ein Standarddatentyp 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

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)

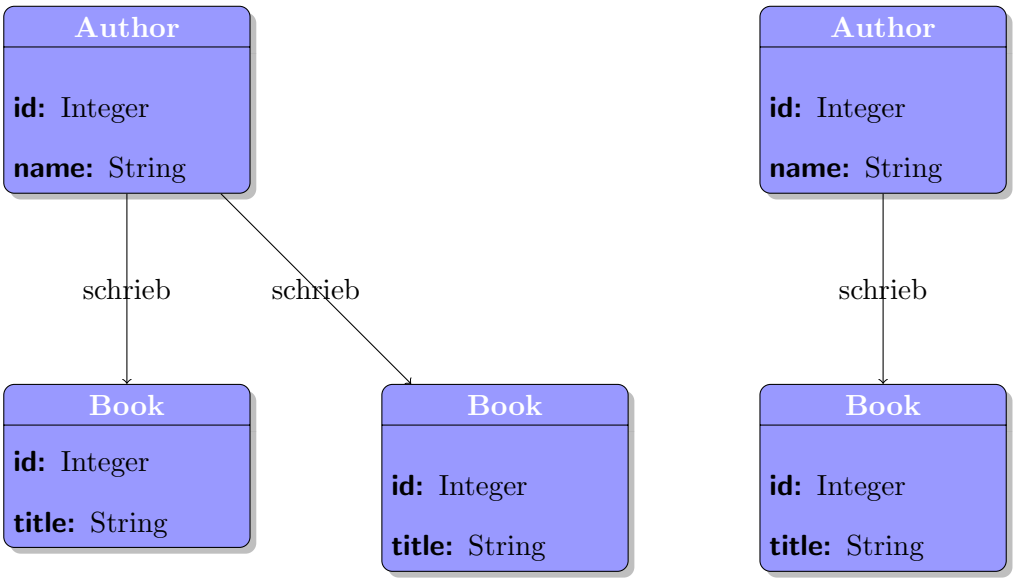


Abbildung 4.1: 3 Bücher, 2 Autoren

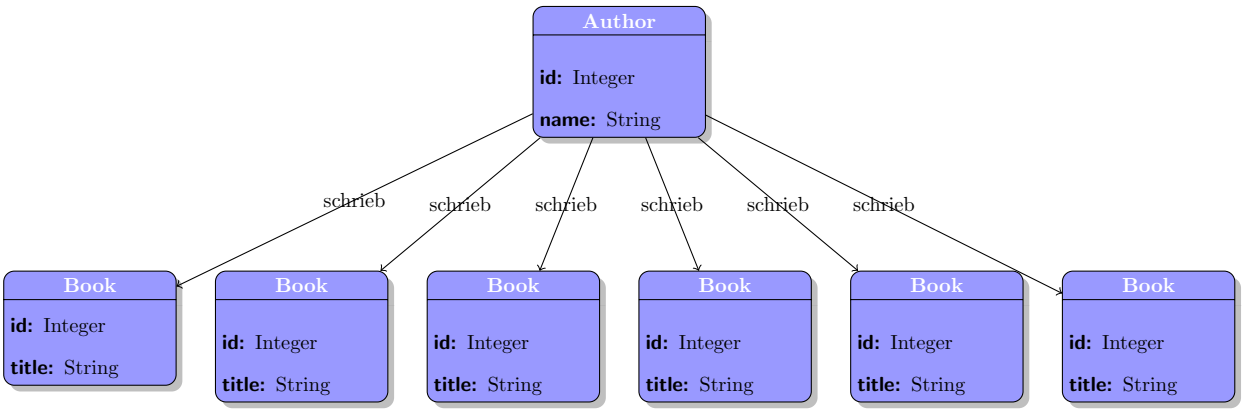


Abbildung 4.2: 6 Bücher, 1 Autor

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 etwaiigen 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 Granularitätseben 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 getest 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 Grapcoverage 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 $E \subseteq V^2 (vgl. Graphlehrbuch) Die Definition muss hierbei erweitert werden mit$:

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 x N. Hierbei ist die Menge als initx x 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 $N_0 start et und in einem Knoten N_f endet, ne$
$1 Kopie$)

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):

```
{
  "kind": "Document",
  "definitions": [
    {
      "kind": "ObjectTypeDefinition",
      "name": {
        "kind": "Name",
        "value": "Book"
      },
      "fields": [
        {
          "kind": "FieldDefinition",
          "name": {
            "kind": "Name",
            "value": "id"
          },
          "type": {
            "kind": "NamedType",
            "name": {
              "kind": "Name",
              "value": "Int"
            }
          }
        },
        {
          "kind": "FieldDefinition",
          "name": {
            "kind": "Name",
            "value": "title"
          },
          "type": {
            "kind": "NamedType",
            "name": {
              "kind": "Name",
              "value": "String"
            }
          }
        },
        {
```

```
          "kind": "FieldDefinition",
          "name": {
            "kind": "Name",
            "value": "author"
          },
          "type": {
            "kind": "NamedType",
            "name": {
              "kind": "Name",
              "value": "Author"
            }
          }
        }
      ]
    },
    {
      "kind": "ObjectTypeDefinition",
      "name": {
        "kind": "Name",
        "value": "Author"
      },
      "fields": [
        {
          "kind": "FieldDefinition",
          "name": {
            "kind": "Name",
            "value": "id"
          },
          "type": {
            "kind": "NamedType",
            "name": {
              "kind": "Name",
              "value": "Int"
            }
          }
        },
        {
          "kind": "FieldDefinition",
          "name": {
            "kind": "Name",
            "value": "name"
          },
          "type": {
            "kind": "NamedType",
```

```
            "name": {
              "kind": "Name",
              "value": "String"
            }
          }
        },
        {
          "kind": "FieldDefinition",
          "name": {
            "kind": "Name",
            "value": "written"
          },
          "type": {
            "kind": "ListType",
            "type": {
              "kind": "NonNullType",
              "type": {
                "kind": "NamedType",
                "name": {
                  "kind": "Name",
                  "value": "Books"
                }
              }
            }
          }
        }
      ]
    }
  ]
}
```


Die Ausgabe in diesem AST verrät uns für jede ÖbjectTypeDefinition"das wir hier einen Knoten des Graphens haben und in dem Fieldsëintrag 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:

FieldDefiniton

NonNullType

ListType

ObjectTypeDefinition

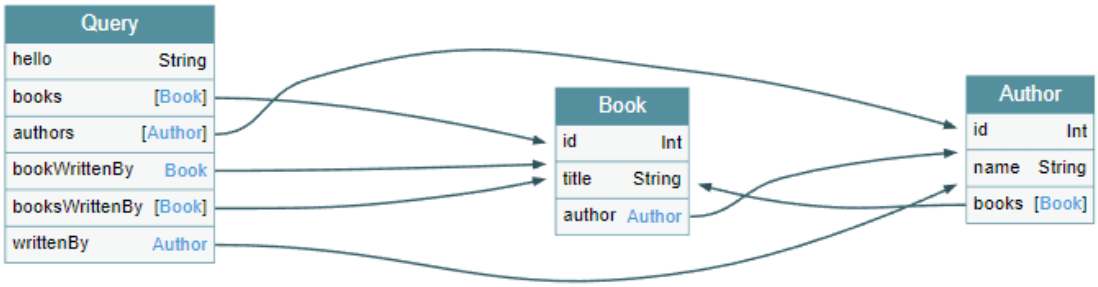
(Quelle X.Y.Z)

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 Äuthoröder "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 →author
- books →author →books
- authors
- authors →books
- authors →books →author
- bookWrittenBy
- bookWrittenBy →author
- bookWrittenBy →author -¿ book
- booksWrittenBy
- booksWrittenBy →author
- booksWrittenBy →author →book
- writtenBy
- writtenBy →book
- writtenBy →book →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 →author →book →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 -> 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 bentuzen um zu zeigen, wie die Testentwicklung für unser Schema aussieht. Jeder gefundene Prime-Path wird nun so untersucht, dass er

7 Testautomatisierung

Hi

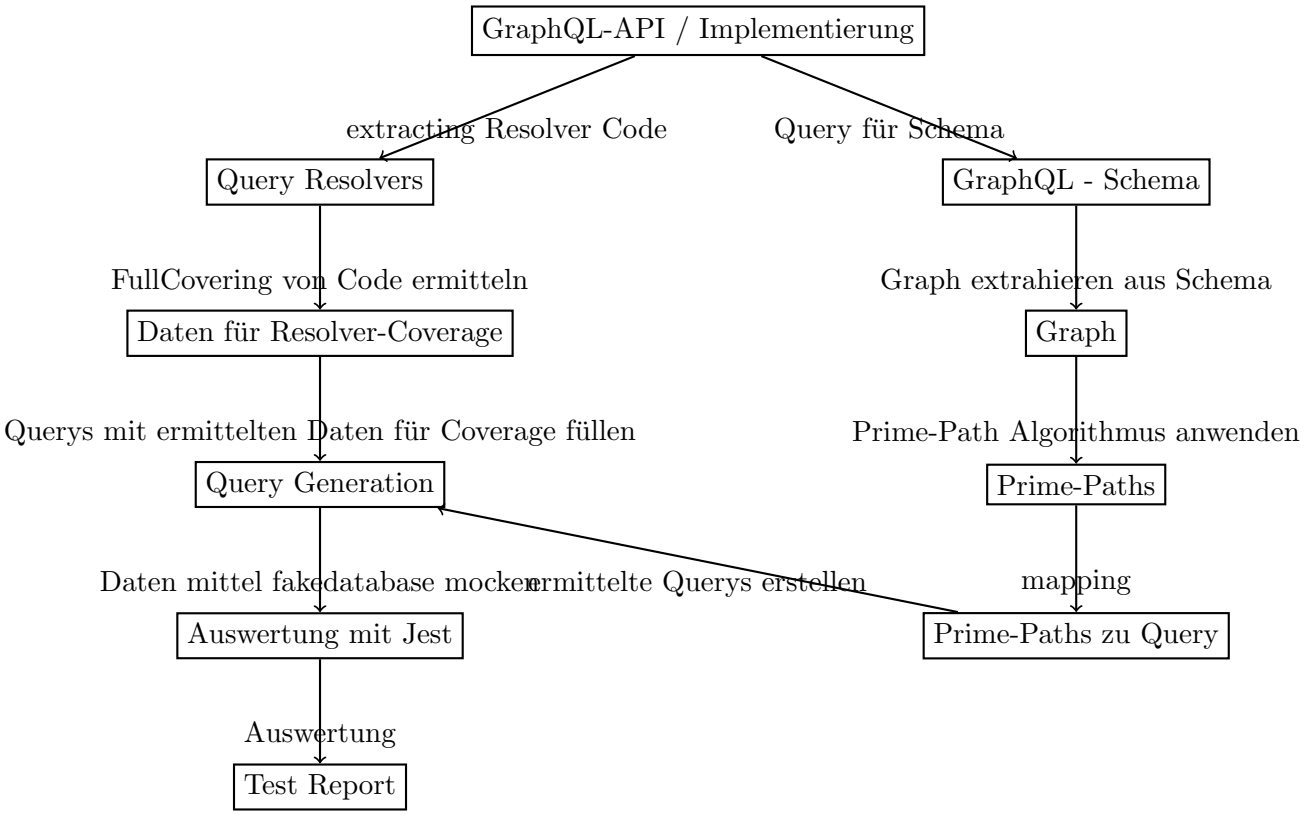
8 Praxis

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 "Baupläneänzulegen 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

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

API

Evolutionärer Algorithmus

Onlineressourcen wurden im Juli 2021 auf ihre Verfügbarkeit hin überprüft.