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

Masterarbeit

Testentwurf & Testautomatisierung für GraphQL

Testdesign & Automation for GraphQL

Tom Lorenz
MatrikelNr.: 3711679
Studiengang: Informatik M.Sc

Datum der Themenausgabe: (hier einfügen)
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

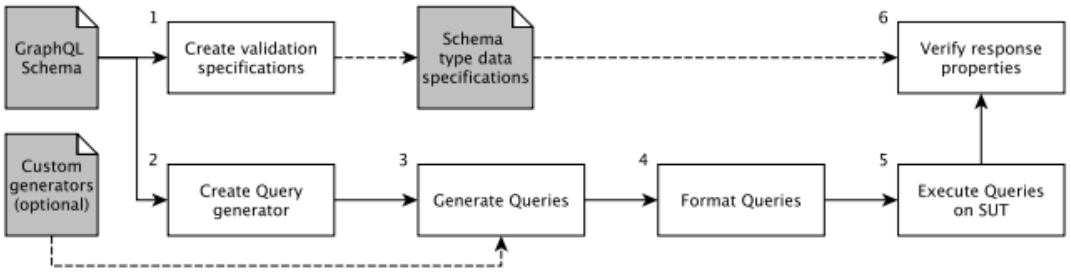
Mit zunehmender Popularität von GraphQL ist es wichtig auch die Qualität von GraphQL-Api's zu validieren und verifizieren. Aktuell gibt es aber noch Einschränkungen an Testtools für GraphQL-Apis, insbesondere im Bereich der automatischen Testanalyse. Im Paper Äutomatic Property-based Testing of GraphQL APIs (Quelle hinzufügen) wurde sich mit einem automatischen Testverfahren für GraphQL-API's beschäftigt. Allerdings bietet diese Arbeit ein Verbesserungspotential, welches in dieser Arbeit untersucht und implementiert werden soll. Im konkreten handelt es sich bei dem automatischen Testverfahren um ein Verfahren, das ein GraphQL-Schema aufgrund seiner Struktur rekursiv untersucht und hieraus tests generiert. Hierbei wird zwar Rücksicht auf die spezielle Graphstruktur genommen allerdings werden spezifische Grapheigenschaften nicht ausgenutzt, um die Tests zu verbessern. Dabei sind diverse Arbeiten publiziert worden die sich mit Graphenüberdeckung beschäftigen welche auch Algorithmen anwenden, die zyklische Strukturen gut überdecken. Ziel dieser Arbeit ist es, dieses Wissen über Graphüberdeckung zu nutzen indem besagte GraphAlgorithmen den rekursiven Suchalgorithmus des Papers ersetzen. Hierdurch erhoffen wir uns eine Verbesserung der Test-Coverage.

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 undendlich großen Testraum. In einem Paper Äutomatic Property-based Testing of GraphQL-API's"(hier Quelle) wurde versucht ein solches automatisches Testtool schon umzusetzen. Allerdings hat dieses Paper zwei große Verbesserungspunkte. Einerseits die Art, wie die Tests generiert werden, andererseits die Auswertung der Tests. Der allgemeine Ablauf des bestehenden Tools ist wie folgt:



Verbesserungen in dieser Arbeit sind insbesondere in den Punkten 2,3 und 6 geplant.

Create Query Generator (Punkt 2) Kapitel Testgenerierung

Generate Queries (Punkt 3) Kapitel Testgenerierung

Verify response properties (Punkt 6) Kapitel Testauswertung

2.1.1 Testgenerierung

Bei der Testgenerierung wurde mit den zyklischen Strukturen in GraphQL Schemas so umgegangen, dass eine Rekursionstiefe definiert wurde, damit man das Problem des unendlichen Testraumes beheben kann. Dieser Ansatz erlaubt es aber leider nicht, dass eine ideale Abdeckung (was das ist wird später genau definiert) gewährleistet werden kann. Insbesondere komplexere, zyklische Strukturen werden hierbei von dem automatischen Tool nicht getestet da die Rekursionstiefe dies oft nicht erlaubt. Mit der Nutzung von graphspezifischen Algorithmen ist es jedoch möglich Graphabdeckungen zu ermitteln auch wenn diese eine zyklische Struktur haben und somit kann algoritmisch das Problem des Papers gelöst werden. In dieser Arbeit sollen diese graphspezifischen Algorithmen implementiert werden und dann mittels Datengeneratoren eigenständig Tests erzeugen.

2.1.2 Testauswertung

Die Auswertung der Tests ist im Paper darauf basierend, dass die zu testende API vor allem auch funktionale Korrektheit überprüft wird, dies bedeutet insbesondere, dass hier die HTTP-Status Codes von Anfragen ausgewertet werden sowie GraphQL eigene Statusmeldungen untersucht werden. Ein richtiger Abgleich im Test findet nicht statt, es wird lediglich überprüft ob der Rückgabe-Datentyp dem erwarteten Datentyp entspricht. Es wäre jedoch besser wenn nicht nur der Rückgabe-Datentyp stimmt sondern auch der Inhalt in diesem Datentyp. In dieser Arbeit soll das Programm mittels eines Orakels verbessert werden. Dies bedeutet, dass die GraphQL API mit Testdaten befüllt wird und Anfragen gestellt werden können, die sich dann auch logisch ineinander auflösen. Hierbei ist insbesondere zu beachten, dass Kreisstrukturen sich in gewisser Weise in sich selbst auflösen, d.h. Eingabeobjekt = Ausgabeobjekt.

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. Wenn der Bezug von Graphentheorie und GraphQL klar ist, wird der eigentliche Algorithmus für die ideale Abdeckung des Graphens algorithmisch erklärt, eine Anwendung auf GraphQL in der Theorie gezeigt und bewiesen. Danach werden die theoretischen Erkenntnisse in einem praktischen Projekt umgesetzt. Hierbei wird ein Tool erstellt, welches aufgrundlage des Überdeckungsalgorithmus & einem Datengenerator Tests erstellt und dann auswertet mittels den alten bzw. erweiterten Ablgeichsmethoden. Um zu zeigen, dass das neue Verfahren eine Verbesserung darstellt wird dann ein Benchmark Test zwischen altem und neuen System erstellt. Bei diesem Benchmark Test werden 3 verschiedene GraphQL-Schemas getestet, 2 Schemas aus dem alten Paper, hier ergibt sich ein direkter Vergleich an z.B. generierten Tests und Graphabdeckung. Im 3ten Schema wird ein speziell sehr zyklisches Schema ausgewertet um zu zeigen, dass einerseits die

Implementierung den Algorithmus korrekt umsetzt und wie groß die Verbesserung in solchen Schemas dann sind.

3 related Work / verwandte Arbeiten

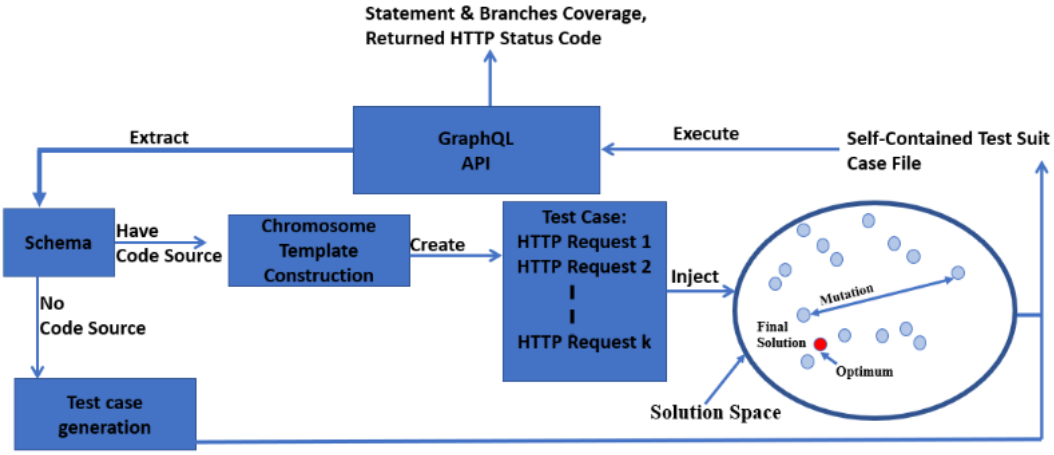
Da Software-Testing ein stetig wachsendes Thema ist und der allgemeine Konsens m̈an kann nicht genug testenëxistiert ist es klar, dass diverse Arbeiten in Richtung Testautomatisierung erstellt wurden. In diesem Abschnitt sollen ähnliche Arbeiten genannt werden sowie Unterschiede zu diesen Arbeiten benannt werden.

3.1 Property Based Testing

In (Paper Quelle hier verlinken) wird der Ansatz des Property-based Testing verfolgt. Property-based Testing ist laut dem Paper heute Synonym mit Random Testing"(vgl. Paper) wobei zufällig hierbei meint, dass Eingabedaten zufällig generiert werden. Die Generierung der eigentlichen Tests geschieht hierbei auf Grundlage des graphenbasierten Schemas (vgl. Paper) indem das Schema rekursiv untersucht wird auf Öbjekteünd Fields"(Objekte = Types; Fields = Felder des Typens). Die generierten Tests werden dann auf dem zu testenden System ausgeführt und ausgewertet. Im Grunde arbeitet dieses Paper ähnlich wie die hier zu untersuchende Methode allerdings nutzen die Autoren bekanntes Wissen nicht vollkommen aus. So ist erwähnt, dass der Query-Generator ein Iterationslimit auf jedem Knoten hat um mögliche zyklische Strukturen zu umgehen. (vgl Creation of the query Generatorïm Paper). Unter anderem dieser Umstand führt dazu, dass unter gewissen Umständen Tests nicht ideal abgebildet werden. Die Annahme, dass ein Iterationslimit nötig ist, ist aber falsch. Es gibt Algorithmen wie z.B. den PrimePath Algorithmus (später mehr dazu) die eine Graphüberdeckung ermitteln können auch in zyklischen Graphen ohne dabei in einer Rekursionsschleife feststecken zu bleiben. Insofern soll der Query-Generation Ansatz vom Property-based Testing"durch unseren neuen Ansatz ersetzt werden. Hierzu jedoch im folgenden dann mehr.

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 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 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.

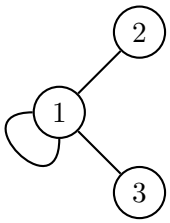
4.1 Graphentheorie

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:

Menge N von Knoten

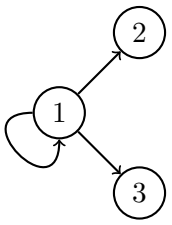
Menge N_0 von Anfangsknoten, wobei $N_0 \subseteq N$

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) entwickelt von Facebook. (vgl. GraphQL-Spec) Ziel dieser Sprache ist es einen möglichst intuitiven und flexiblen Ansatz für Datenkommunikation zu bieten. Hierbei ist GraphQL für die Strukturierung und Anfragesteuerung zuständig. GraphQL selbst ist keine Programmiersprache sondern verteilt die Anfragen entsprechend der definierten Spezifikation in GraphQL. Wie man GraphQL spezifiziert und was Ünter der Haube"dabei passiert wird im folgendem erklärt. Da GraphQL ein äußerst detaillierter Standard ist, werden hier nur grundlegende Konzepte betrachtet die für die Arbeit wichtig sind, insbesondere wichtige Aspekte für die Testgenerierung.

4.2.1 Schema & Typen

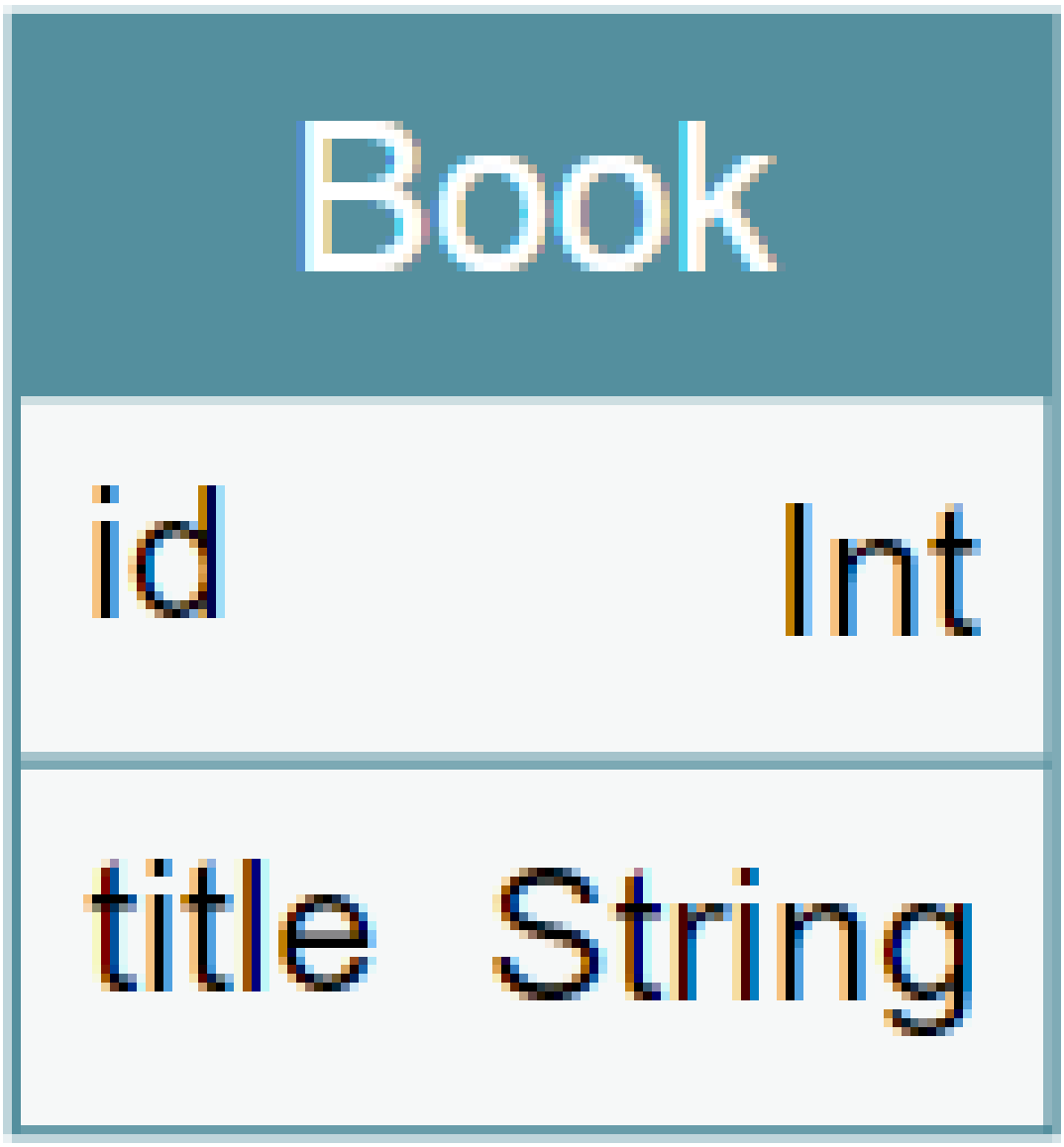
Test

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.

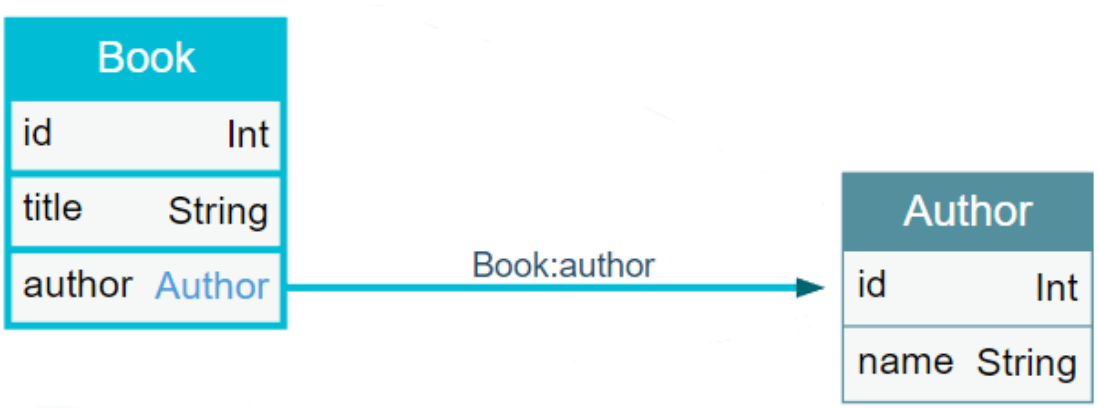


Ein Objekt enthält 1 bis n Propertys. Diese Property kann entweder ein Standarddatentyp sein oder auf einen Type verweisen, dies kann der eigene Type oder auch ein anderer Type sein. Fügen wir unserem Beispiel des Buches eine Property hinzu mit dem Type Author wobei der Author selbst wie folgt definiert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

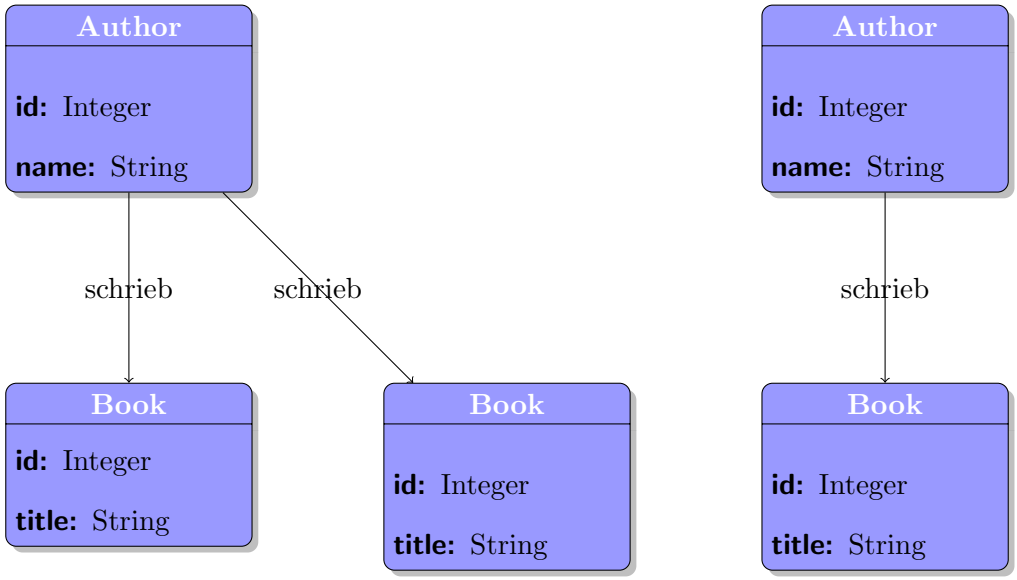


Abbildung 4.1: 3 Bücher, 2 Autoren

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)

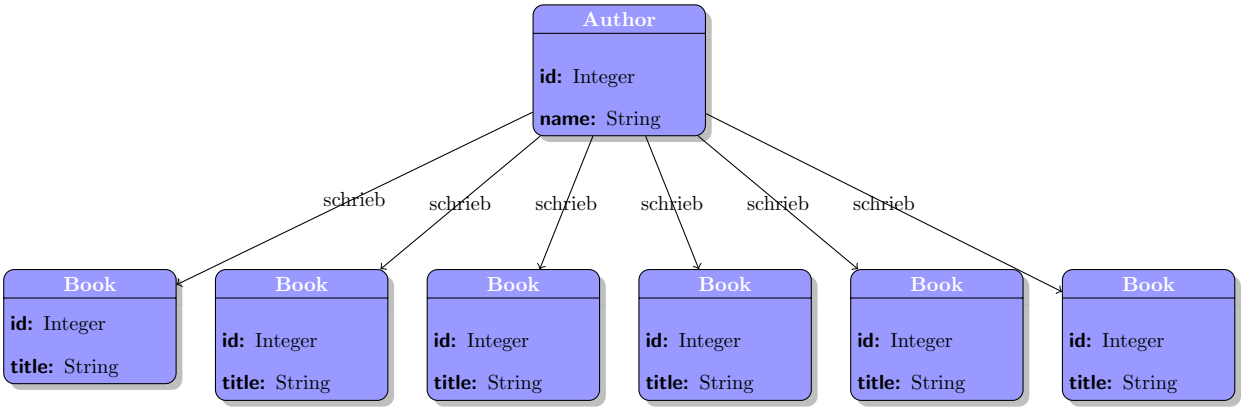


Abbildung 4.2: 6 Bücher, 1 Autor

4.4 Testen

(Hier Leen nach Buchempfehlungen fragen)

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 um einen Beweis zu haben, dass die Software das tut was vorgegeben ist.

4.4.1 Arten von Tests

Acceptance Testing

System Testing

Integration Testing

Module Testing

Unit Testing

4.4.2 Test-Coverage

4.4.3 Test-Coverage Graphen

4.4.4 Graphcoverage Graphcoverage Kriterien

Edge Coverage

Node Coverage

Edge-Pair Coverage

Prime-Path Coverage

4.4.5 Graphcoverage für Code

4.4.6 Graphcoverage für GraphQL

4.4.7 Prime-Path Coverage Algorithmus

5 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 ...

5.1 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.

5.1.1 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.

5.1.2 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

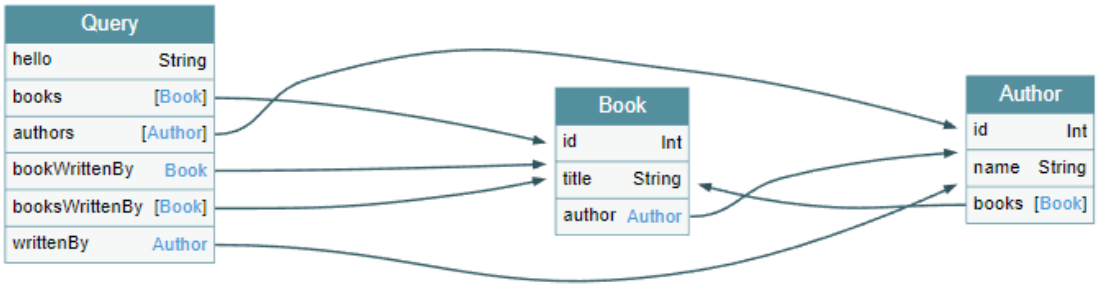
(Quelle X.Y.Z)

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.

5.1.3 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.

5.2 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

6 Testautomatisierung

Hi

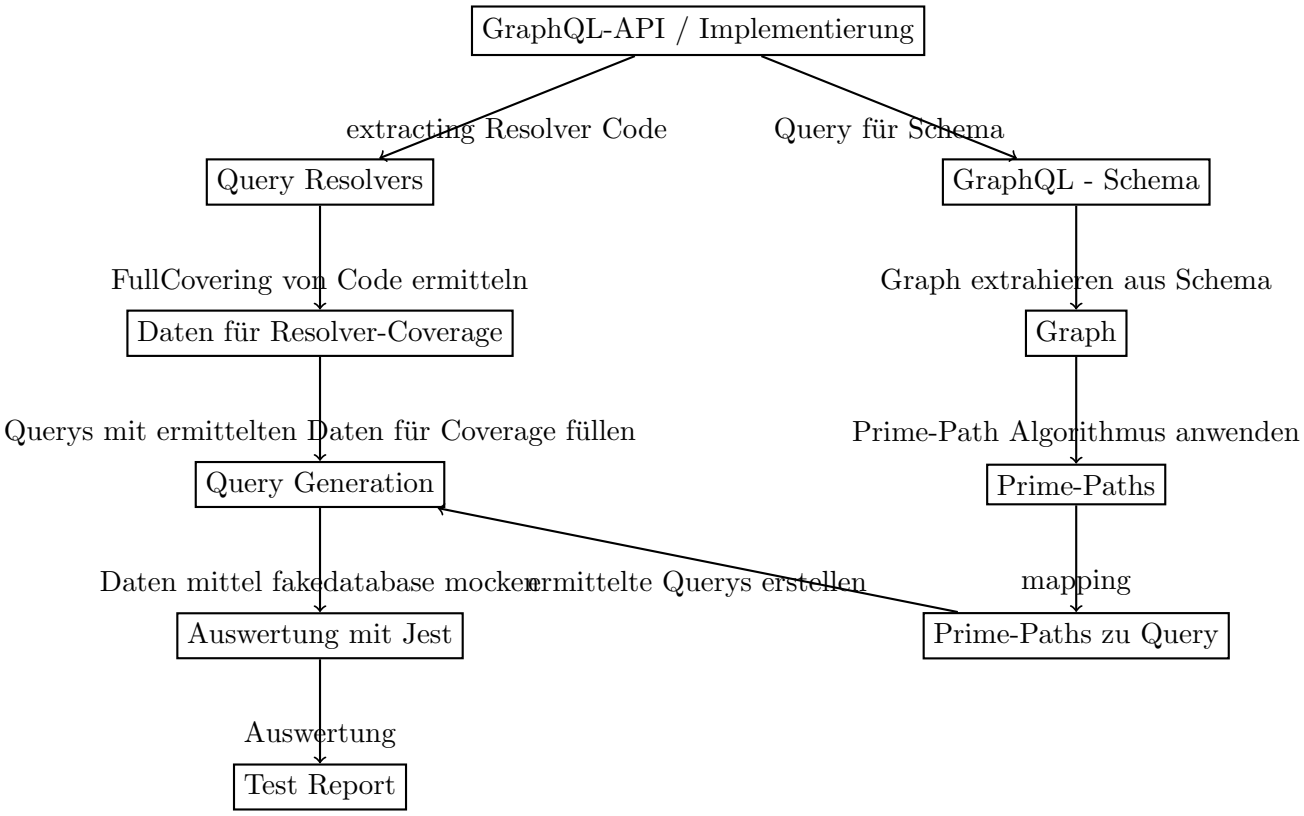
7 Praxis

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.

7.1 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.

7.2 Ablauf der Generierung



7.3 Requirements an das Tool

8 zukünftige Arbeit

Hi

9 Fazit

Hi

10 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.