Technology
Arts Sciences
TH Köln

TH Köln

Praxisprojekt

Wintersemester 2018/2019

Praxisprojekt

Autor:
Elisha Witte
In Kooperation mit:
antwerpes ag

Betreuer:
Christian Kohls
Christian Noss
Christiane Grünloh
Volker Schäfer

10. Februar 2019

Inhaltsverzeichnis

1 Einleitung

1.1 Nutzungsproblem

Im Unternehmen *antwerpes ag* wird, zunehmend auch teamübergreifend, mit REST APIs gearbeitet. Applikationen bieten Schnittstellen an, die von Entwicklern, teilweise in anderen Teams, konsumiert werden. Dabei entstehen typische Kollaborationsprobleme: nicht ausreichende bzw. schlechte Dokumentation und Inkonsistenzen in den Definitionen der Schnittstellen, da sich diese insbesondere während der Entwicklungszeit häufig ändern.
Gleichzeitig werden dabei viele Aufgaben der Entwicklung und Interaktion mit den APIs (bspw. Validierung von Anfragen, Testing der Endpunkte oder Konfiguration von Tools wie Postman) vernachlässigt oder manuell ausgeführt.
Dies führt im besten Fall zu mehr Zeitaufwand und Code-Duplizierung, im schlimmsten Fall zu weiteren Inkonsistenzen, welche dann manuell korrigiert werden müssen.

1.2 Zielsetzung

Unterstützend für eine gute Entwicklung ist ein Entwicklungsprozess, der das Projekt in allen Punkten, die einheitlich organisiert und durchgeführt werden sollten, festlegt; sowie Techniken verwendet, welche die Tätigkeiten möglichst gut unterstützen (Jochen Ludewig und Horst Lichter, 2007: 220).
Es soll daher ein System entworfen und entwickelt werden, welches ein einheitliches Konzept zum Arbeiten mit APIs einführt.
Ziel des Projektes ist die Optimierung des Design- und Entwicklungsprozesses von auf HTTP basierenden REST APIs durch Verwendung einer API Spezifikation als *single source of truth* für Dokumentation und Testing. Dadurch können einige manuelle Aufgaben automatisiert werden, und viele Inkonsistenzen behoben werden. Insgesamt steigt die Testabdeckung und Qualität der Dokumentation der Schnittstellen, womit die Kollaboration zwischen Entwicklern verbessert wird. Das resultierende System soll dabei mit möglichst geringem Aufwand in bereits bestehende Projekte integriert werden können.

2 Einführung in die Thematik

Innerhalb dieses in die Thematik einführenden Kapitels werden Grundlagen bezüglich der Bereiche, in denen sich das Projekt bewegt, aufgezeigt. Zunächst wird ein kurzer Überblick über die grundlegenden Prinzipien von HTTP und REST gegeben. Anschließend wird untersucht, wie APIs mithilfe von verschiedenen Deskriptionsformaten maschinenlesbar beschrieben werden können. Ziel dieser Untersuchung ist es, eine Grundlage für die abstrakte Modellierung von REST APIs zu schaffen, die verwendet werden kann, um eine API Spezifikation in eine interne Datenstruktur zu überführen, welche dann anschließend angereichert wird, um automatisiert Artefakte wie Dokumentation und Test-Cases zu generieren. Im Zuge dessen wird im letzten Abschnitt analysiert, welche Möglichkeiten in diesem Feld bereits existieren.

2.1 HTTP

Das *Hypertext Transfer Protocol* (HTTP) ist ein Anwendungsprotokoll zur Kommunikation über ein Netzwerk. HTTP ist das primäre Kommunikationsprotokoll im World Wide Web, und wurde von der Internet Engineering Task Force (IETF) und dem World Wide Web Consortium (W3C) in einer Reihe von RFC-Dokumenten (z.B. RFC 7230, RFC 7231 und RFC 7540) standardisiert. In diesem Abschnitt werden die wichtigsten Charakteristiken aus den aufgeführten RFCs zusammengefasst.

Eine HTTP-Nachricht wird in der Regel über TCP übertragen, und besteht aus 4 Komponenten:

Verb / Methode: Operationstyp, der durchgeführt werden soll, beispielsweise das Anfragen einer Ressource.

Ressourcenpfad: Ein Bezeichner, der angibt, auf welche Ressource die HTTP-Operation angewendet werden soll.

Headers: Zusätzliche Metadaten, ausgedrückt als Liste von Key/Value-Paaren.

Body: Enhält die Nutzdaten der Nachricht.

Das HTTP Protokoll erlaubt die folgenden Operationen bzw. Methoden:

GET: Die angegebene Ressource soll im Body-Teil der Antwort zurückgegeben werden.

HEAD: Analog zu *GET*, nur dass die Nutzdaten nicht zurückgegeben werden sollen. Dies ist nützlich, wenn nur überprüft wird, ob eine Ressource existiert, oder wenn nur die Header abgerufen werden sollen.

POST: Daten werden an den Server geschickt. Häufig wird diese Methode benutzt, um neue Ressourcen anzulegen.

DELETE: Die angegebene Ressource löschen.

PUT: Die angegebene Ressource mit neuen Daten ersetzen.

PATCH: Ändert eine Ressource, ohne diese vollständig zu ersetzen.

TRACE: Liefert die Anfrage so zurück, wie der Server sie empfangen hat.

OPTIONS: Liefert eine Liste der vom Server unterstützten Methoden auf einer Ressource.

CONNECT: Eine Tunneling-Verbindung über einen HTTP-Proxy herstellen, die normalerweise für verschlüsselte Kommunikationen benötigt wird.

Wenn ein Client eine HTTP-Anfrage sendet, schickt der Server eine HTTP-Antwort mit Headern und möglicherweise Nutzdaten im Body zurück. Darüber hinaus enthält die Antwort auch einen numerischen, dreistelligen Statuscode. Es gibt fünf Gruppen von Statuscodes, die durch die erste Ziffer identifiziert werden können:

1xx: Wird für informierende Antworten verwendet, wie z.B. der Meldung, dass die Bearbeitung der Anfrage trotz der Rückmeldung noch andauert.

2xx: Wird zurückgegeben, wenn die Anfrage erfolgreich bearbeitet wurde. Der Server könnte beispielsweise weiter spezifizieren, dass eine neue Ressource angelegt wurde (201), z.B. durch einen POST-Befehl, oder dass im Antwortkörper nichts erwartet wird (204), z.B. durch einen DELETE-Befehl.

3xx: Wird für Umleitungen benutzt. So kann dem Client bspw. mitgeteilt werden, dass sich eine Ressource an einem neuen Ort befindet.

4xx: Bei der Bearbeitung der Anfrage ist ein Fehler aufgetreten, der durch den Client verursacht wurde. Dies können beispielsweise falsch formatierte Anfragen (400), Anfragen die vom Server nicht bearbeitet werden können (422), oder auch Anfragen auf nicht existierende Ressourcen (404) sein.

5xx: Bei der Bearbeitung der Anfrage ist ein Fehler aufgetreten, dessen Ursache beim Server liegt.

2.2 REST

Representational State Transfer ist ein Muster von Ressourcenoperationen, das sich als Industriestandard für das Design von Webanwendungen etabliert hat (Battle und Benson, 2008: 2). Während der traditionelle SOAP-basierte Ansatz für Web Services vollwertige Remote-Objekte mit Remote-Methodenaufruf und gekapselter Funktionalität verwendet, beschäftigt sich REST nur mit Datenstrukturen und der Übertragung ihres Zustands.

Representational State Transfer (REST) ist ein Softwarearchitekturstil, der die architektonischen Prinzipien, Eigenschaften und Einschränkungen für die Umsetzung von internetbasierten verteilten Systemen definiert (Fielding und Taylor, 2000: 86). REST basiert auf fünf Kernprinzipien (Tilkov u. a., 2015: 11):

- Ressourcen als Abstraktion von Informationen, identifiziert durch einen eindeutigen *resource identifier* (die URI).

- Verknüpfungen / Hypermedia (HATEOAS). Jede Repräsentation einer Ressource sollte ebenfalls Links zu anderen relevanten Ressourcen enthalten. Da in der Praxis die Verwendung von HATEOAS jedoch recht selten ist (Rodriguez u. a., 2016: 35–36) werden im Folgenden auch solche APIs als RESTful bezeichnet, die dieses Prinzip nicht umsetzen.

- Verwendung von Standardmethoden. Der Zugriff auf Ressourcen und deren Manipulation erfolgt mit den in HTTP definierten Standardmethoden. Jede Methode hat ihr eigenes standardisiertes Verhalten und erwartete Statuscodes.

- Unterschiedliche Repräsentationen. Clients kennen nicht direkt das interne Format und den Zustand der Ressourcen; sie arbeiten mit Ressourcendarstellungen (z.B. JSON oder XML), die den aktuellen oder beabsichtigten Zustand einer Ressource darstellen. Die Deklaration von Inhaltstypen in den Headern von HTTP-Nachrichten ermöglicht es Clients und Servern, Darstellungen korrekt zu verarbeiten (Rodriguez u. a., 2016: 23).

- Statuslose Kommunikation. Interaktionen zwischen einem Client und einer API sind zustandslos, d.h. jede Anfrage enthält alle notwendigen Informationen, die von der API zur Verarbeitung benötigt werden; es wird kein Zustand auf dem Server gespeichert.

2.3 Spezifikationsformate

In den letzten Jahren wurde viele Vorschläge gemacht, wie REST APIs menschen- und maschinenlesbar beschrieben werden können. Allen gemeinsam ist, dass die grundlegenden Eigenschaften von REST und HTTP (Ressourcen, Operationen, Requests, Responses, Parameter, usw.) in einem Modell zusammengefasst werden. Hierfür existieren eine Vielzahl von *Description Languages* (DLs), von denen in diesem Projekt OpenAPI, die am weitesten verbreitete (Scherer, 2016: 38), als Basis benutzt wird.

2.3.1 JSON Schema

Aufgrund seiner leichten Verständlichkeit für Menschen wie auch Maschinen hat sich JSON zum beliebtesten Format entwickelt, um API-Anfragen und Antworten über das HTTP-Protokoll zu senden (Pezoa u. a., 2016: 263). Durch diese Popularität ist allerdings auch der Bedarf gestiegen, mithilfe eines Schemas den Aufbau eines JSON Dokuments konkret zu definieren. Dies kann z.B. für die Validierung von Anfragen oder Antworten dienen. Ohne Integritätsschicht müssen viele Fälle berücksichtigt werden, die bei fehlerhaften API-Aufrufen auftreten, was vermieden werden kann, wenn eine Schemadefinition verwendet wird, um Dokumente herauszufiltern, die nicht die richtige Form haben.

JSON Schema ist eine einfache Schemasprache, die es Benutzern ermöglicht, die Struktur von JSON-Dokumenten einzuschränken. Die Definition ist noch lange kein Standard (die Spezifikation befindet sich derzeit im siebten Entwurf), aber es gibt bereits eine wachsende Anzahl von Anwendungen, die JSON-Schema Definitionen unterstützen, und eine Vielzahl von Tools und Paketen, die die Validierung von Dokumenten anhand von JSON-Schema ermöglichen (Pezoa u. a., 2016: 264). Dies kann selbst zur Client-seitigen Validierung von Benutzereingaben verwendet werden (Sturgeon, 2018c).

JSON Schema stellt mehrere Schlüsselwörter zur Beschreibung und Validierung von JSON Dokumenten zur Verfügung. Beispielsweise kann so der Typ von Daten festgelegt werden (`null`, `boolean`, `object`, `array`, `number`, `integer` oder `string`). Es können ebenfalls vielzählige Restriktionen zur Validierung definiert werden, unter anderem: `enum`, `minimum`, `maximum`, `minLength`, `maxLength`, `regex` (Validierung anhand eines RegEx-Musters), `required`, `email`, `date-time` und `uri`.

2.3.2 OpenAPI

Die OpenAPI DL ist Teil des Swagger Frameworks, einem Open-Source-Framework, das aus einem JSON- bzw. YAML-Format zur Beschreibung von

RESTful APIs und aus diversen Werkzeugen, die daraus zum Beispiel eine Dokumentation generieren können, besteht. OpenAPI und JSON Schema sind zwei verschiedene Spezifikationen, die sich gegenseitig inspiriert haben, sich aber subtil unterscheiden.

OpenAPI zielt darauf ab, sowohl das Servicemodell zu beschreiben (die API im Allgemeinen, Endpunkte, Anfragemetadaten wie Header, Authentifizierungsstrategien, Antwortmetadaten usw.), als auch den HTTP-Request/Response-Body mit einem Bündel von Schlüsselwörtern basierend auf dem JSON-Schema, die im Laufe der Zeit divergiert haben. Es gibt aktuell Bemühungen, diese Divergenz zu lösen (Sturgeon, 2018b). JSON Schema wiederum zielt darauf ab, eine Instanz von JSON-Daten zu beschreiben, wie sie in einer HTTP-Anfrage oder -Antwort gefunden wird, ist aber in keiner Weise auf eine HTTP-API beschränkt (Sturgeon, 2018a).

Die OpenAPI DL ist ähnlich wie andere Spezifikationsformate strukturiert: zunächst werden allgemeine Informationen zur API, wie Name, Version, Pfad und Authentifizierungsverfahren festgelegt; im Anschluss folgen die Definitionen der einzelnen Endpunkte, mit ihren Parametern und Requests bzw. Responses.

2.4 Testing von REST APIs

In diesem Abschnitt wird ein kurzer Überblick über die wichtigsten Testverfahren gegeben, mit denen REST APIs getestet werden können:

2.4.1 Funktionstests

Die wichtigste testbare Eigenschaft eines Programms ist seine Funktionalität, die durch die Anforderungen spezifiziert ist (Jochen Ludewig und Horst Lichter, 2007: 457). Funktionstests sind im Wesentlichen ein Test spezifischer Funktionen innerhalb der Codebasis. Diese Funktionen wiederum stellen spezifische Szenarien dar, um sicherzustellen, dass die API innerhalb der erwarteten Parameter funktioniert und dass Fehler gut behandelt werden, wenn die Ergebnisse außerhalb der erwarteten Parameter liegen. In der grundlegends-

ten Form sollte ein korrekter Response-Body die Darstellung der angeforderten Ressource unter Verwendung des angeforderten Medientyps oder mindestens einer gültigen HTTP-Fehlercodemeldung enthalten. Funktionstests sollten nicht nur den regulären Testfall einbeziehen, sondern auch Szenarien von Errata- und Edge-Cases in das Testverfahren implementieren. Ein Beispiel hierfür ist eine Anfrage mit falschen Attributen oder Attributwerten, auf die die API mit einem korrekten Fehlercode antworten sollte. Eine bei APIs besonders wichtige Unterkategorie der Funktionstests sind die Regressionstests, mit denen sichergestellt werden kann, dass bei internen Änderungen die Rückgaben der API weiterhin mit dem definierten Vertrag (bspw. der Spezifikation) kompatibel sind, und somit Client-Anwendungen, die auf die API angewiesen sind, weiterhin wie bisher funktionieren.

2.4.2 Last- und Stresstests

Hierbei wird getestet, ob sich das System auch unter hoher oder höchster Belastung so verhält, wie es gefordert war (Jochen Ludewig und Horst Lichter, 2007: 458). Diese Tests werden typischerweise nach Fertigstellung der Codebasis durchgeführt. REST APIs werden gegen den theoretischen regulären Datenverkehr, den die API im normalen, täglichen Gebrauch erwartet, getestet. Ein zweiter Belastungstest wird typischerweise mit dem theoretischen maximalen Traffic durchgeführt. Eventuell wird auch ein dritter Belastungstest mit Traffic, der über dem theoretischen Maximum liegt, durchgeführt.

2.4.3 Sicherheitstests

Sicherheitstests und Penetrationstests sind Teil des Sicherheitsauditprozesses. Bei Sicherheitstests wird zum Beispiel die Verwaltung von Benutzerrechten und die Validierung von Berechtigungsprüfungen für den Ressourcenzugriff getestet. Die Penetrationstests gehen einen Schritt weiter. Bei dieser Art von Test wird die API von jemandem angegriffen, der nur über begrenzte Kenntnisse der API selbst verfügt, um den Bedrohungsvektor aus einer externen Perspektive zu beurteilen.

2.5 Verwandte Arbeiten und Systeme

Zweck dieses Unterkapitels ist die Darstellung von verwandten Arbeiten, welche sich ebenfalls mit der Automatisierung von Testing und Dokumentation von REST APIs beschäftigen. Daneben werden ebenfalls bereits existierende oder im Stadium der Entwicklung befindende Systeme einbezogen. Die betrachteten Systeme und Arbeiten werden mit der Zielsetzung des vorliegenden Projektes verglichen und Unterschiede und Gemeinsamkeiten hervorgehoben.

2.5.1 Wissenschaftliche Arbeiten

In einem Paper aus 2017 haben Ed-douibi, Izquierdo und Cabot (2018) ein System zur automatischen Generierung von Testcases auf Basis einer OpenAPI Spezifikation entworfen und als Prototyp implementiert. Es wird dafür zunächst die Spezifikation in ein OpenAPI Metamodell extrahiert, welches dann in mehreren Schritten mit Beispielwerten für Parameter und Attribute angereichert und anschließend in ein Testsuite Metamodell umgewandelt wird. Hieraus werden im letzten Schritt die entsprechenden Testcases generiert. Berücksichtigt werden hierbei nicht nur nominale sondern auch fehlerhafte Tests (also HTTP-Anfragen mit bewusst falschen Parameter- oder Attributwerten, mit denen auf entsprechende Validierungsfehler geprüft wird). Das System kommt der in diesem Projekt gesetzten Zielsetzung und dem dafür gewählten Vorgehen sehr nahe, es gibt jedoch einige Einschränkungen, über die sich dieses Projekt abgrenzen kann:

- Das System ist auf ein bestimmtes Spezifikationsformat (OpenAPI) beschränkt, und nicht mit anderen Formaten kompatibel. Es wurde zwar ein Metamodell erstellt, in welches die OpenAPI Spezifikation übertragen wird (und somit über entsprechende Erweiterungen eventuell auch andere Formate), bei der Erstellung dieses Modells wurden jedoch nur die Charakteristiken der OpenAPI Spezifikation berücksichtigt. Ein System welches mehrere Formate unterstützt, sollte bereits bei der Modellierung entsprechende Analysen und Abstraktionen anwenden.

- Der Prototyp ist lediglich als Eclipse-Plugin mit Generierung der Testcases in Java implementiert worden, und somit auf eine bestimmte Entwicklungsumgebung und Sprache (Java) beschränkt. Das Ziel dieses Projektes ist die Implementierung einer CLI-Applikation, welche so modular gestaltet ist, dass mehrere Input- und Output-Formate möglich sind. Aufgrund der Vorgaben im Unternehmen wird für die Implementierung des Prototyps als Input eine OpenAPI Spezifikation und als Output PHP gewählt.

In einem Paper aus 2015 stellten Fertig und Braun (2015) einen modellbasierten Ansatz zur Generierung von Testcases vor. Hierzu wurde eine eigene Description Language erstellt, mit der die REST API beschrieben werden kann. Der Fokus liegt bei der Generierung von möglichst vielen Testcases; für jeden Endpunkt werden alle möglichen Kombinationen von Headern und Attributen abgebildet. Es werden zwar auch hier zum Teil fehlerhafte Tests generiert, mit dem Ziel zu überprüfen, ob der Server fehlerhafte Angaben korrekt abweist, jedoch sind diese im Vergleich zum oben vorgestellten System sehr beschränkt, und hängen stark davon ab, dass in der Spezifikation komplette Definitionen vorhanden sind. Ebenso scheinen optionale Query-Parameter nicht unterstützt zu werden. Die Implementierung des Prototyps wurde ebenfalls für Java entwickelt. Die größte Einschränkung ist jedoch die Verwendung einer eigenen DL, wodurch zusätzliche Einstiegshürden entstehen, und Tools für bestehende Spezifikationsformate nicht verwendet werden können.

Pinheiro, Endo und Simao (2013) verwenden einen ähnlichen Ansatz, basieren ihr Modell jedoch auf einer angepassten *UML protocol state machine*, welche vom Tester manuell erstellt werden muss. Die Umsetzung fand hier jedoch nur im Ansatz statt, und wichtige Aspekte wie Ressourcenrepräsentationen oder optionale Parameter wurden nicht implementiert.

Chakrabarti und Kumar (2009) entwickelten ein HTTP-Testing-Tool, das speziell für das Testen von RESTful-Webservices entwickelt wurde. Es basiert auf XML-Dateien, welche die erforderlichen Testfälle konfigurieren. In einem Pilotprojekt wurden Testfälle zur automatischen Generierung

erstellt, die einer vollständigen Liste aller gültigen Kombinationen von Query-Parameter-Werten entsprechen. Daraus ergab sich eine Testsuite mit 42.767 Testfällen. Ein wesentlicher Nachteil des Systems ist das benötigte Vorwissen zur Konfiguration der Testcases mithilfe der zum Teil recht komplexen XML-Definitionen.

Ein etwas komplexerer Ansatz wurde von Arcuri (2017) vorgestellt. Hier wird ebenfalls von einer OpenAPI bzw. Swagger Definition ausgegangen, und auf Basis dieser Tests generiert. Mithilfe von Bytecode-Manipulation bei JVM-Sprachen werden Informationen zu Code-Coverage und Verzweigungsabständen abgerufen, mit denen die optimalen Testcases ausgewählt werden. Abgesehen von der offensichtlichen Einschränkung auf JVM-Sprachen, prüft das System auch lediglich auf HTTP Statuscodes, und gleicht die erhaltenen Antworten nicht mit Schemas o.ä. ab.

Ein Großteil der Arbeit in der Literatur konzentriert sich auf Black-Box-Tests von SOAP-Webservices, die mit WSDL (Web Services Description Language) beschrieben werden, welche jedoch inzwischen kaum noch verwendet wird (Scherer, 2016: 37–39). Es wurden verschiedene Strategien vorgeschlagen, einige Beispiele sind Xu, Offutt und Luo (2005), Bai u. a. (2005), Martin, Basu und Xie (2006), Ma u. a. (2008) und Bartolini u. a. (2009). Diese werden jedoch aufgrund der unterschiedlichen Architektur und den damit geltenden Rahmenbedingungen nicht weiter betrachtet, und nur der Vollständigkeit halber aufgeführt.

2.5.2 Vorhandene Softwarelösungen

Zusätzlich zu den wissenschaftlichen Arbeiten gibt es auch viele bereits bestehende Softwarelösungen bzw. Tools, die mit API Spezifikationen arbeiten und bestimmte Aufgaben automatisieren können. Es sei erwähnt, dass zusätzlich zu den hier aufgelisteten Open Source Projekten auch noch einige kommerzielle, kostenpflichtige Lösungen verfügbar sind (z.B. Stoplight.io oder Readme.io), welche hier jedoch aufgrund der Vorgaben des Projektes seitens des Unternehmens nicht weiter betrachtet werden.

Dokumentation

ReDoc ist ein Tool, mit dem auf Basis einer OpenAPI Spezifikation eine moderne, responsive HTML API Dokumentation generiert werden kann. Über ein Plugin werden ebenfalls Code-Beispiele in verschiedenen Sprachen unterstützt. Die Anpassung des Designs ist zwar grundsätzlich möglich, jedoch nicht dokumentiert. Nicht unterstützt werden ausführbare Beispielanfragen.
Swagger UI ist Teil des Swagger Frameworks, und ermöglicht ähnlich wie ReDoc das Generieren einer HTML API Dokumentation anhand einer OpenAPI Spezifikation. Anders als bei ReDoc sind hier ausführbare Beispielanfragen möglich, jedoch fehlen jegliche Anpassmöglichkeiten des Themes und Code-Beispiele.
Aglio und Snowboard sind zwei Dokumentations-Tools für das API Blueprint Spezifikationsformat, welche jedoch schwer anpassbar sind.
Weitere nennenswerte Tools sind Spectacle und DapperDox, mit ähnlichen Einschränkungen.

Testing

Zum Testen von APIs mithilfe einer API Spezifikation ist besonders Dredd zu erwähnen, ein sprachunabhängiges CLI-Tool zur Validierung von API-Spezifikationen gegen die Backend-Implementierung der API. Hierzu wird die API Spezifikation (im OpenAPI oder API Blueprint Format) eingelesen, und für jeden in der Beschreibung definierten Endpunkt eine Anfrage an den Server geschickt. Dafür werden alle definierten Parameter und Attribute mit ihren entsprechenden Beispielwerten benutzt. Die Antworten des Servers werden anschließend validiert; wird entweder ein nicht erfolgreicher Statuscode zurückgegeben, oder die Antwort entspricht nicht dem in der Spezifikation definierten Format schlägt der Test fehl. Dredd ist besonders hilfreich um zu testen, dass die API Spezifikation (und somit auch die Dokumentation) auf dem aktuellen Stand ist. Das eigentliche Testen der API Implementierung ist jedoch sehr eingeschränkt, da für jeden Endpunkt lediglich eine Anfrage ausgeführt wird (und somit eventuelle Query-Parameter nicht berücksichtigt werden), und ebenfalls keine fehlerhaften Anfragen abgeschickt werden.

2.6 Abgrenzung von anderen Arbeiten

Die vorliegende Arbeit hat das Ziel eine Gesamtlösung zur Vereinheitlichung und Optimierung des API-Entwicklungsprozesses zu entwerfen und zu implementieren. Zu Teilen dieses Konzeptes gibt es bereits, wie im vorherigen Abschnitt vorgestellt, einige Softwarelösungen. Im Folgenden werden kurz die Alleinstellungsmerkmale vorgestellt, die während der Analyse aufgefallen sind.

Testing
Zum Testing von REST APIs auf Basis einer API Spezifikation sind zwar einige wissenschaftliche Arbeiten zu finden, welche jedoch meist nur als Prototyp implementiert sind, und umfangreiche Anpassungen bzw. Erweiterungen benötigen würden um den Anforderungen gerecht zu werden. Einige Punkte, die von den untersuchten Systemen nicht abgedeckt werden, sind:

- Generierung von Testcases in einer anderen Sprache als Java, in dem Fall dieses Projektes primär in PHP.
- Unterstützung von mehr als einem API-Spezifikationsformat. Die meisten Arbeiten basieren auf einer OpenAPI Spezifikation, verwendet werden in der Industrie jedoch auch andere (bspw. API Blueprint oder RAML). Hier müssten zusätzliche Analysen und Überlegungen durchgeführt werden, wie mehrere Formate in ein gemeinsames, internes Datenmodell überführt werden können, und welche Transformationen dafür eventuell nötig sind.
- Modularer Aufbau, zur einfachen Erweiterung. Insbesondere beim Generieren der Testfälle ist es sehr wahrscheinlich, dass langfristig verschiedene Output-Formate unterstützt werden müssen. Das entwickelte Tool sollte also so aufgebaut sein, dass mit geringem Aufwand Testcases für andere Sprachen oder Frameworks generiert werden können (durch Entwicklung einer Erweiterung welche leicht in das Tool eingebunden werden kann).

Von *Dredd* kann sich das Projekt alleine durch die höhere Abdeckung der Testcases (optionale Query-Parameter und fehlerhafte Anfragen) abgrenzen.

Dokumentation

Zur Generierung von Dokumentation wurde eine Vielzahl von Tools gefunden, die bereits einen Großteil der Anforderungen abdecken. Dennoch wurde bei den Open-Source Projekten keine Lösung gefunden, welche eine API Dokumentation *auf Basis mehrerer Spezifikationsformate* generiert, die responsiv, interaktiv (durch ausführbare Beispielanfragen) und leicht anpassbar ist, sowie automatisch generierte Code Snippets enthält.

3 Anforderungen, Ziele und Rahmenbedingungen

3.1 Empirische Untersuchung

Zu Beginn der Anforderungsermittlung wurde eine Umfrage im Unternehmen durchgeführt, bei der die Entwickler gebeten wurden, Auskunft über die Projekte zu geben, die sie betreuen. Ziel der Untersuchung war es, mehr über die verwendeten Technologien und damit verbundenen Rahmenbedingungen zu erfahren, sowie persönliche Einschätzungen der Entwickler zu Testing und anderen automatisierbaren Aspekten der API-Entwicklung einzuholen. In diesem Abschnitt werden die Ergebnisse der Umfrage aufgeführt.

3.1.1 Verwendete Technologien

Alle befragten Entwickler gaben an, dass sie API-Projekte betreuen, die in PHP geschrieben sind. Andere Sprachen sind dabei kaum vertreten (dargestellt in Abbildung 1). Als primär verwendete Frameworks wurden Symfony (60%) und Laravel (30%) genannt, wobei auch einige andere Frameworks zu einem geringeren Maß verwendet werden (siehe Abbildung 2).

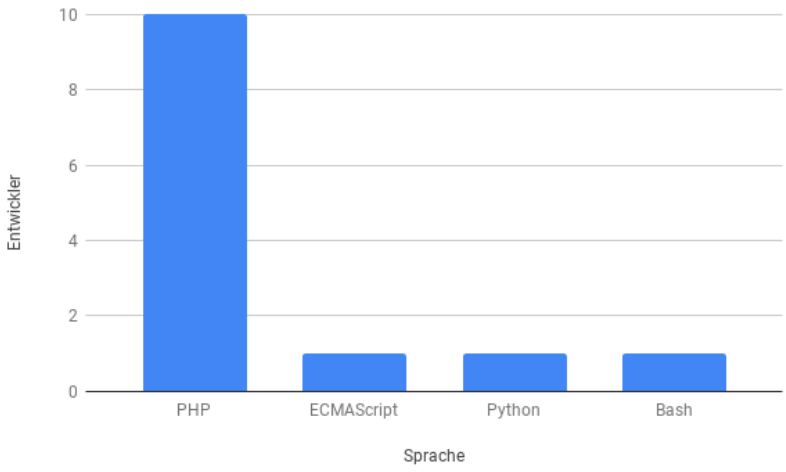

Abbildung 1: Verwendete Sprachen

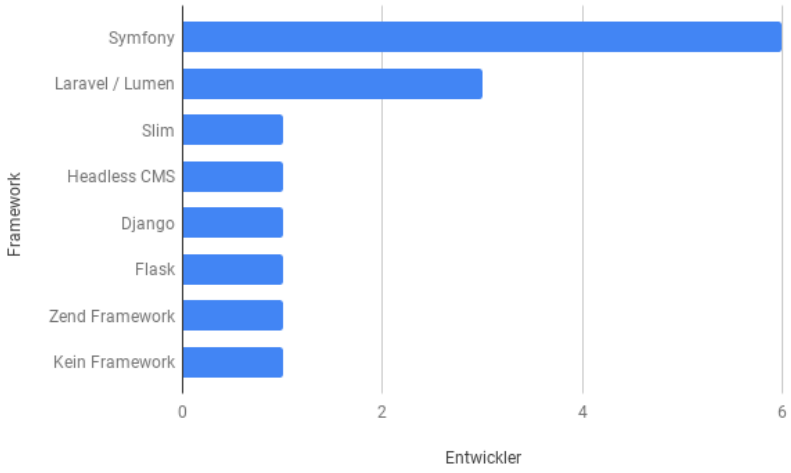

Abbildung 2: Verwendete Frameworks

90% der Entwickler gaben an, dass ihre APIs primär die REST-Architektur befolgen; die restlichen 10% benutzen GraphQL als Grundlage. Auf die Frage, ob sie Description Languages zur Beschreibung ihrer APIs verwenden, antworteten 50% der Entwickler positiv. Die Beschreibungen werden dabei zu 60% manuell geschrieben, und zu 40% automatisch aus dem Code der Anwendung generiert. Die verwendeten Description Languages sind in Abbildung 3 dargestellt.

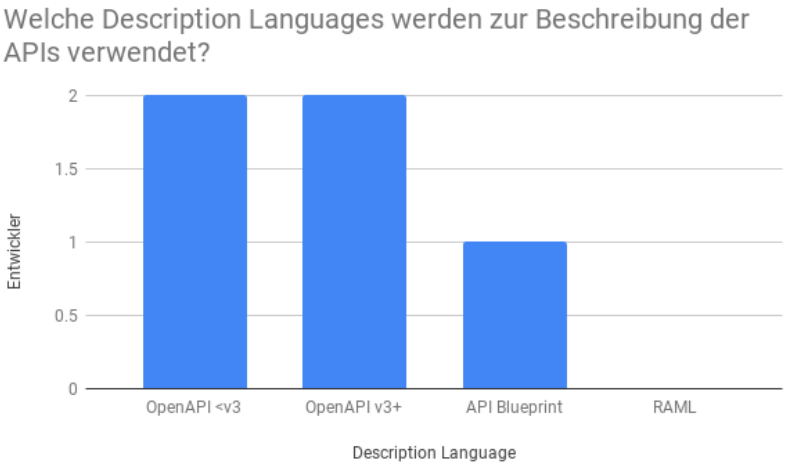

Abbildung 3: Verwendete Description Languages

3.1.2 Testing der APIs

Bei der Frage wie die APIs aktuell getestet werden, fällt auf, dass viele Entwickler keine automatisierten Tests verwenden. Stattdessen werden häufig nur manuelle Tests, insbesondere mithilfe von Postman, durchgeführt. Lediglich 30% der Entwickler gaben an, dass sie Unit bzw. Integration Tests einsetzen (siehe Abbildung 4). Dennoch bewerteten die meisten Entwickler die Testabdeckung der APIs insgesamt als eher positiv, wenn auch hervorzuheben ist, dass einige APIs eine unzureichende bzw. nichtexistente Testabdeckung aufweisen (siehe Abbildungen 5 und 6).
Es wurde ebenfalls gefragt, wie hilfreich verschiedene Automatisierungen für die Entwickler wären. Herauszuheben sind:

- 60% aller Entwickler würden die automatische Generierung von grundlegenden Integreation Tests für ihre Projekte als sehr hilfreich einstufen, 30% als eher hilfreich, und lediglich 10% haben eine neutrale Einstellung.
- 45,5% würden abrufbare, aktuelle JSON Schemas der API Endpunkte als eher hilfreich einstufen, 27,3% als sehr hilfreich, 27,3% sind neutral.

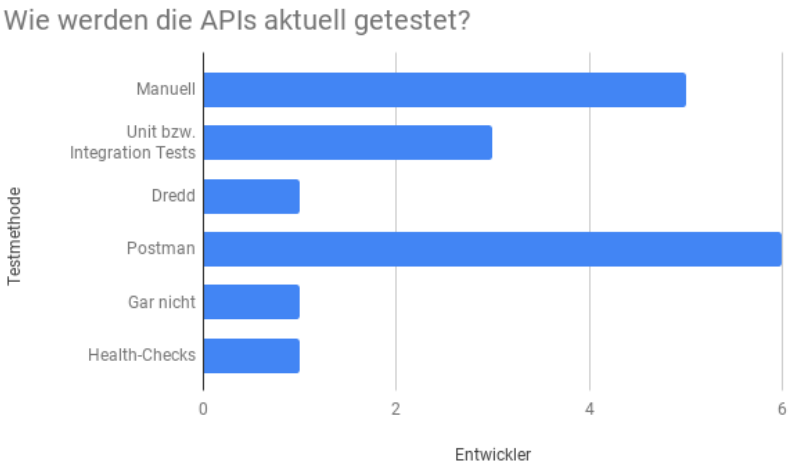

Abbildung 4: Testing der APIs

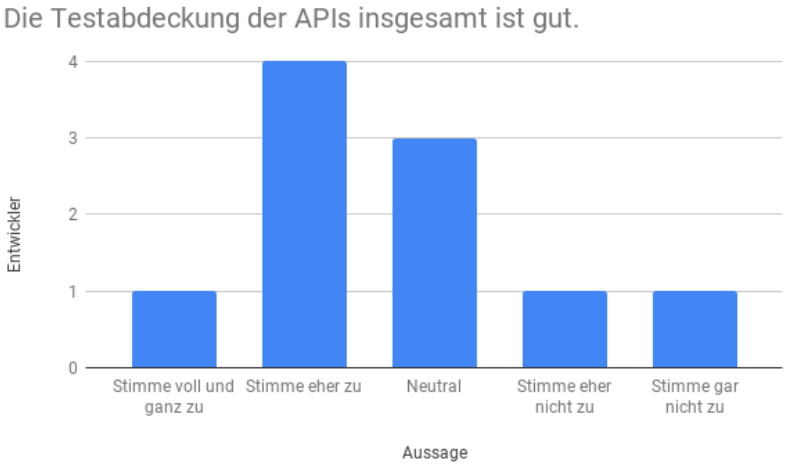

Abbildung 5: Testabdeckung der APIs insgesamt

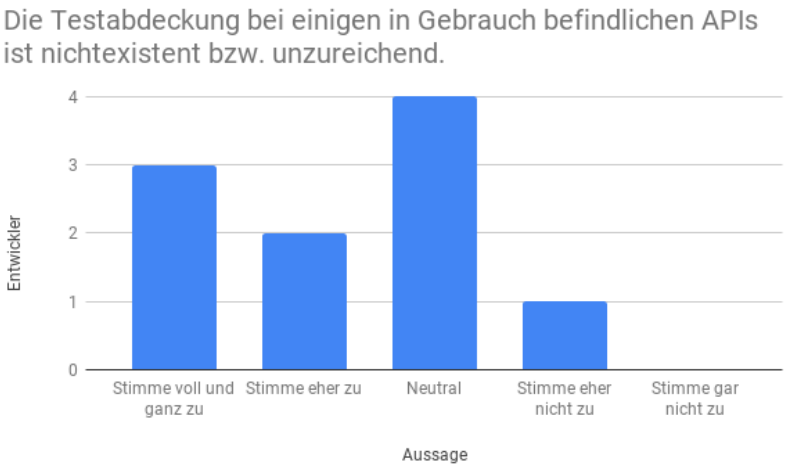

Abbildung 6: Unzureichende Testabdeckung mancher APIs

3.2 Ziele

Im Folgenden werden die konkreten Ziele des Systems hierarchisch angeordnet. Zunächst werden die strategischen, langfristigen Ziele festgelegt, aus denen sich anschließend die taktischen und operativen Ziele ableiten. Die Ziele sind in der jeweiligen Kategorie nach ihrer Priorität angeordnet. Bei

allen Zielen wird davon ausgegangen, dass eine API Spezifikation vorliegt, auf Basis derer die Aufgaben durchgeführt werden können.

3.2.1 Strategische Ziele

1. **Optimierung des REST-API Entwicklungsprozesses.**
   Durch zumindest teilweise Automatisierung von verschiedenen Aufgaben im Entwicklungsprozess (bspw. Testing) soll Zeit eingespart werden.

2. **Erhöhte Softwarequalität.**
   Die Vereinheitlichung des Prozesses und auch die automatisierte Generierung von Artefakten sollte zu einer höheren Softwarequalität und einfacheren Wartung führen. Durch die Verwendung von generierten Testcases soll insbesondere bei Projekten mit zur Zeit nicht vorhandenen Tests eine zufriedenstellende Testabdeckung erreicht werden. Auch bei neuen Projekten soll die Anwendung eine gewisse Grundarbeit leisten, und unnötige manuelle Arbeit abnehmen.

3. **Vereinheitlichung des REST-API Entwicklungsprozesses.**
   Mit der Unterstützung des Entwicklungsprozesses durch das System sollen Grenzen und Vorgaben gesetzt werden, in denen sich die Entwickler bewegen. Das Design der REST APIs folgt einem einheitlicherem Konzept, wodurch Nutzer nicht bei jeder API mit neuen Designentscheidungen konfrontiert sind. Generierte Artefakte sind konsistent, und ihre Nutzung muss nur einmal erlernt werden.

3.2.2 Taktische Ziele

1. **Abstraktes Datenmodell.**
   Das Datenmodell des Systems sollte so konzipiert sein, dass mehrere Spezifikationsformate überführt werden können.

2. **Erweiterbarkeit des Systems.**
   Das System sollte so modular konzipiert und implementiert sein, dass Teile des Systems erweiterbar sind. Damit könnten beispielsweise neue

Parser (zur Unterstützung von neuen Spezifikationsformaten) oder Factories (z.B. zur Generierung von Testcases in einer anderen Sprache) in das System eingebunden werden.

3. **Generierung von Testcases.**
   Es sollen automatisch sinnvolle Testcases generiert werden, um die Funktionalität der API zu testen. Diese sollten dabei unterstützend wirken. Das Ziel ist es nicht, die Entwickler komplett vom Schreiben von Tests zu entbinden; vielmehr sollen einfache Tests zwar komplett automatisiert werden, bei solchen Tests, die noch manuelle Ergänzungen der Entwickler benötigen, soll das Systems allerdings nur hilfreiche Vorarbeit leisten.

4. **Generierung von Dokumentationsartefakten.**
   Zur Verbesserung der Dokumentation der API sollen automatisch kollaborationsunterstützende Artefakte generiert werden.

5. **Überprüfung auf Best Practices.**
   Es sollte (optional) auf die Befolgung von Best Practices beim Design von REST-APIs geprüft werden sollen (Linting).

3.2.3 Operative Ziele

1. **Generierung von JSON Schemas.**
   Aus dem eingelesenen Metamodell sollen JSON Schemas generiert werden, welche unter anderem für Testcases verwendet werden.

2. **Generierung von nominalen Testfalldefinitionen.**
   Zur Überprüfung der Funktionalität der API, sollen Testcases generiert werden, die jeden Endpunkt der API mit den in der Spezifikation definierten Parametern und Attributen aufrufen, und auf valide Antworten prüfen.

3. **Generierung von fehlerhaften Testfalldefinitionen.**
   Es sollen nicht nur Testfälle generiert werden, die auf Funktionalität der API in Einhaltung aller in der Spezifikation definierten Einschränkungen prüfen, sondern auch solche, die auf korrekte Fehlerbe-

handlung prüfen. Dies kann z.B. in Form einer Anfrage mit falschen Attributen oder Attributwerten geschehen.

4. **Generierung einer API Dokumentation**.
   Es soll eine responsive, interaktive und leicht anpassbare API-Dokumentation generiert werden, welche die Benutzung der API verdeutlicht.

5. **Generierung und Synchronisation von Postman Collections.**
   Als weiteres kollaborationsunterstützendes Artefakt sollen automatisch Postman Collections generiert werden, und diese automatisch über die Postman API bei allen Entwicklern synchronisert werden.

6. **Einfache Einbindung in Projekte.**
   Das System sollte ohne großen Aufwand in neue oder bestehende Projekte eingebunden werden können.

7. **Partielle Überschreibung von Testcases bei Neugenerierung.**
   Insbesondere bei den Testcases, die noch weitere manuelle Ergänzungen der Entwickler benötigen, sollen bei Neugenerierung nicht die Ergänzungen überschrieben werden.

8. **Unterstützung von semantischen Tests.**
   Da die automatisch generierten Testcases lediglich die Syntax, und nicht die Semantik der API testen können, sollte das System die Entwickler dabei unterstützen Tests zu schreiben, die auf die Semantik prüfen. Davon ausgehend, dass Entwickler die Semantik der API mit replizierbaren Testdaten bereits manuell geprüft haben, könnte dies zum Beispiel durch einen neuen Befehl umgesetzt werden, der die aktuellen Antworten der API zu jeder möglichen Parameterkombination speichert, und daraus ein Testskelett generiert.

3.3 Anforderungen

Aus der Auswertung der Umfrage, sowie weiteren Gesprächen mit Entwicklern des Unternehmens, lassen sich folgende funktionale Anforderungen ableiten, angeordnet nach ihrer Priorität:

1. Die Anwendung muss eine OpenAPI Spezifikation einlesen und verarbeiten können. Andere Spezifikationsformate werden zu einem geringeren Maße verwendet, sollten aber zukünftig auch unterstützt werden.

2. Für alle API Endpunkte müssen valide JSON Schemas sowohl für Requests wie auch Responses generiert werden.

3. Für die 2 meistverwendeten Frameworks (Symfony und Laravel) müssen grundlegende Testcases generiert werden, die auf valide Antworten des Servers prüfen.

4. Es sollte zu jedem API Endpunkt mindestens ein Testcase generiert werden. Für jede mögliche Parameterkombination sollte ebenfalls ein neuer Testcase generiert werden.

5. Die Testcases sollten die Antworten des Servers mithilfe der generierten JSON Schemas validieren.

6. Um die Anfragen in den Testcases ausführen zu können, müssen für erforderliche Parameter und Attribute Beispielwerte abgeleitet werden.

7. Es sollten nicht nur nominale, sondern auch fehlerhafte Testcases generiert werden. Für jedes Attribut, für das ein fehlerhafter Wert abgeleitet werden kann, sollte ein neuer Testcase erstellt werden, und überprüft werden, dass der Server einen Client-Fehler zurückgibt.

8. Die Anwendung sollte so strukturiert sein, dass einzelne Teile leicht ausgetauscht werden können, und untereinander kompatibel sind. Hierfür werden Interfaces definiert, welche die einzelnen Komponenten implementieren.

9. Relevante Optionen, die sich eventuell von Projekt zu Projekt unterscheiden, wie die genaue Zusammensetzung der einzelnen Komponenten, oder auch Dateipfade der generierten Artefakte, sollten mithilfe einer Konfigurationsdatei angepasst werden können.

10. Die Anwendung kann Dokumentationsartefakte wie eine HTML API-Dokumentation oder Postman Collections generieren.

11. Postman Collections sollten in einem CI-Kontext automatisch über die Postman API synchronisiert werden.

12. Die Dokumentation sollte responsiv und interaktiv sein. Die Interaktivität bezeichnet hierbei die Möglichkeit Beispielanfragen direkt aus der Dokumentation heraus auszuführen.

3.4 Funktionsumfang des Prototyps

In dem Prototypen, der im Zuge dieses Projektes entstehen soll, sollen die Ziele 1, 2 und 3 umgesetzt werden. Der Fokus liegt somit auf dem Testing von APIs. Die konkreten Anforderungen hierzu sind die oben aufgelisteten Nummern 1-7. Im Prototyp wird jedoch nur ein Eingabe-Format (eine OpenAPI-Spezifikation), und ein Ausgabe-Format für die Testcases (Laravel) implementiert. Der Prototyp soll zeigen, dass es möglich ist auf Basis einer API Spezifikation automatisch grundlegende Integration Tests für alle Endpunkte der API zu generieren. Dadurch werden nicht nur verschiedene Aspekte der Anwendung getestet, sondern auch sichergestellt, dass der Vertrag der API, der in der Spezifikation festgelegt ist, nicht gebrochen wird. Durch die automatische Generierung von Testcases kann hier eine sehr hohe Abdeckung des Vertrages (d.h. alle Endpunkte und Parameterkombinationen, mit Validierung der jeweiligen Antworten mithilfe der in der Spezifikation definierten Schemas) erreicht werden.

4 Konzeption des Prototyps

4.1 Datenmodellierung

Die zentrale Datenstruktur der Applikation ist die Beschreibung der API selbst, welche als Grundlage für alle Operationen verwendet wird. Da sich der Prototyp beim Eingabeformat auf das OpenAPI Spezifikationsformat beschränkt, wurde zunächst ein OpenAPI Metamodell erstellt, welches in Abbildung 7 dargestellt ist. Dieses Modell wird von der Applikation als Basis-Datenstruktur verwendet. Zur Unterstützung mehrerer Eingabeformate müssten in einer späteren Iteration auch andere Spezifikationsformate untersucht werden, um Gemeinsamkeiten abzuleiten und ein Datenmodell zu erstellen, in das alle Formate überführt werden können.

Es wurden ebenfalls zwei weitere Datenmodelle erstellt, die für die im Prototyp festgelegten Ziele benötigt werden. Zunächst wurde ein Modell der JSON-Schema Spezifikation erstellt, welche sich nur in wenigen Divergenzen von der OpenAPI Schema Definition unterscheidet:

1. Einige im OpenAPI Schema vorhandene Attribute wie `nullable` oder `deprecated` werden nicht unterstützt.
2. Die `type` Angabe kann sowohl ein String wie auch ein Array von Strings sein.
3. In `$schema` wird zusätzlich ein Link zur verwendeten JSON-Schema Version angegeben.

Diese Unterschiede müssen bei einer Umwandlung rekursiv aufgelöst werden, da ein Schema weitere verschachtelte Schema-Defitionen enthalten kann. Das Modell einer JSON-Schema Definition ist in Abbildung 8 dargestellt.

Das Testsuite-Metamodell (Abbildung 9) stellt den Aufbau einer Testsuite dar, welche aus mehreren Testcases besteht. In jedem Testcase wird eine Anfrage mit entsprechenden Parametern ausgeführt, sowie eine Reihe von Assertions.

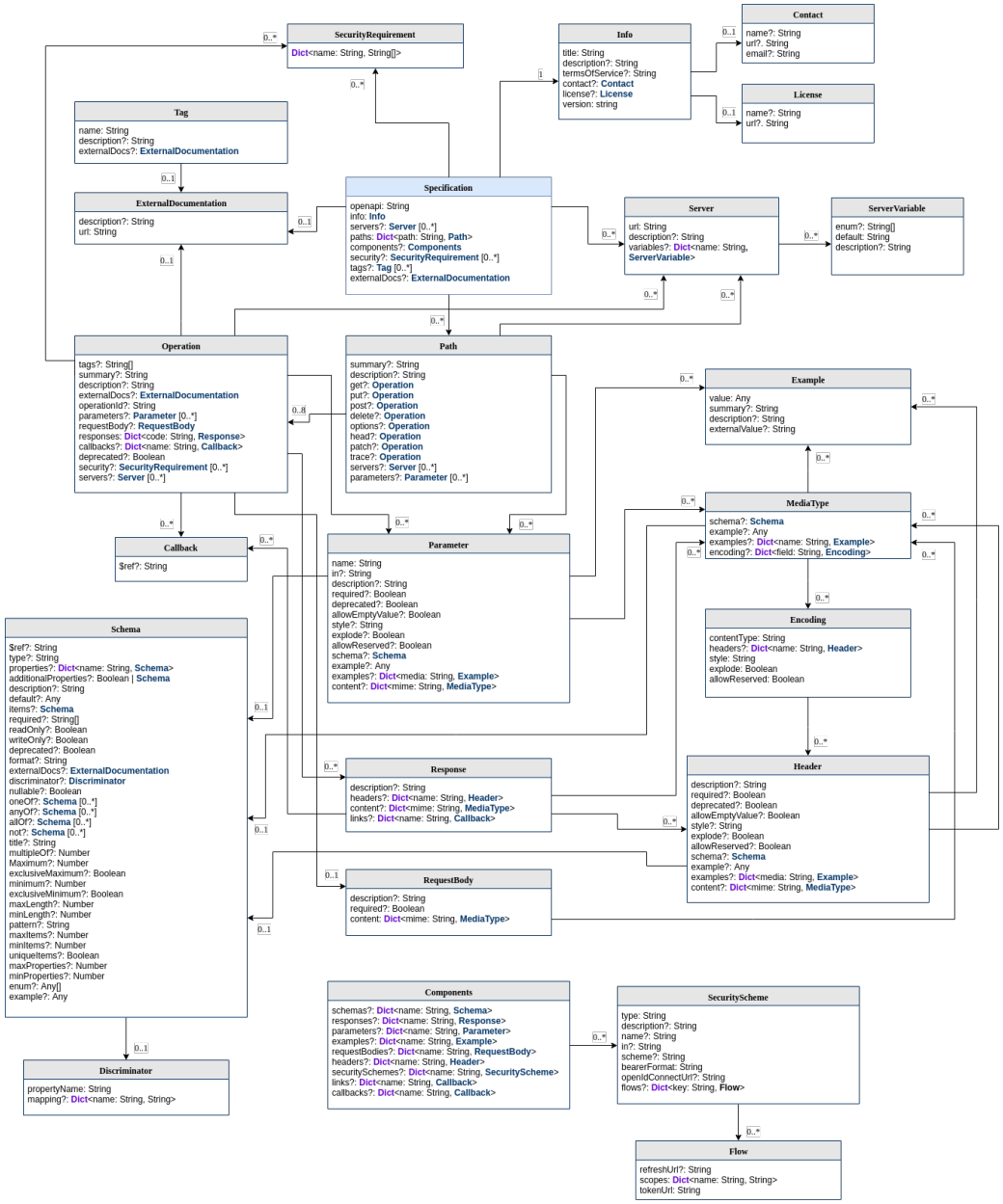

Abbildung 7: OpenAPI Metamodell

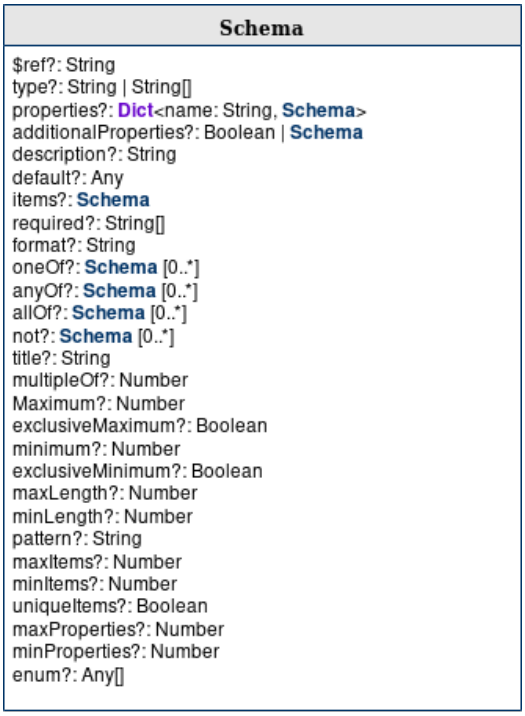

Abbildung 8: JSON-Schema Metamodell

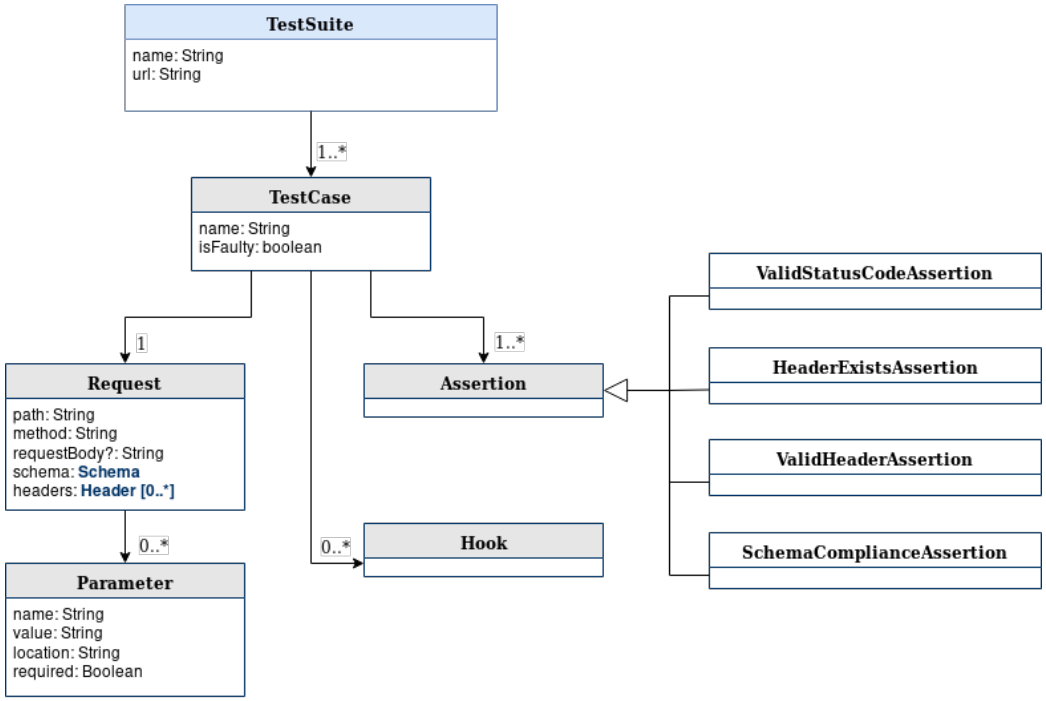

Abbildung 9: Testsuite Metamodell

1. `ValidStatusCodeAssertion` prüft, ob der HTTP-Statuscode der erhaltenen Antwort valide ist. Bei nominalen Testcases wird ein erfolgreicher Statuscode (2xx) erwartet, bei fehlerhaften Testcases ein Client-Fehler (4xx).

2. `HeaderExistsAssertion` prüft, ob alle in der Spezifikation definierten HTTP-Header vorhanden sind.

3. `ValidHeaderAssertion` prüft, ob alle HTTP-Header valide Werte haben.

4. `SchemaComplianceAssertion` validiert die vom Server erhaltene Antwort mit dem passenden JSON-Schema.

Ein Testcase führt ebenfalls mehrere Hooks aus, die von dem Entwickler angepasst werden können. Dadurch kann zum Beispiel Authentifizierung ermöglicht werden.

4.2 Architektur

Unter Berücksichtigung der beiden Ziele **Modularität** und **einfache Einbindung in Projekte** wird für die Umsetzung eine Javascript CLI-Applikation entwickelt. In den meisten Projekten sind bereits Javascript-Abhängigkeiten installiert, sodass sich das Tool leicht einbinden lässt. Durch NPM und die Aufteilung der Applikation in einzelne Pakete mit dynamischen Imports lässt sich ebenfalls die Modularität erreichen.

Die CLI-Applikation besteht aus einer Menge an Befehlen, welche alle den gleichen Aufbau haben, schematisch dargestellt in Abbildung 10. Jeder Befehl besteht aus einem *Parser*, der die API Spezifikation einliest, einem oder mehreren *Decorators*, welche die geparste Spezifikation mit befehlsspezifischer Logik und Informationen anreichern, sowie einer oder mehreren *Factories*, die einen Output generieren.

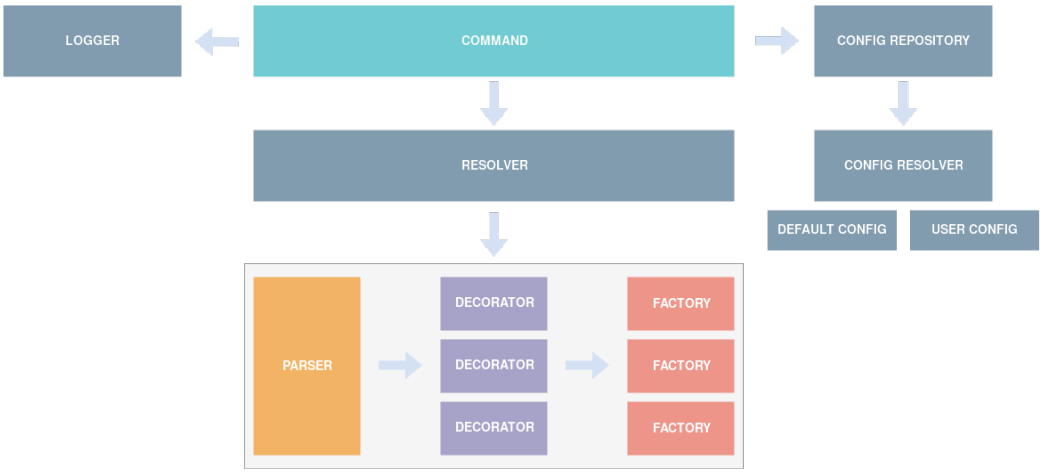

Abbildung 10: Architektur

Diese Komponenten sind beliebig austauschbar und konfigurierbar. Dadurch wird ermöglicht, dass bei Änderungen am Input- oder Output-Format die restliche Logik nicht verändert werden muss. So können z.B. mithilfe von 2 Factories Tests für zwei unterschiedliche Sprachen oder Frameworks generiert

werden, ohne dass etwas an dem Parsing oder Anreichern der Daten (im Falle der Tests z.B. das Ableiten von Parameterwerten) verändert werden muss. Durch dynamische Imports mithilfe eines *Resolvers* werden die einzelnen Pakete zur Laufzeit eingebunden. Somit können Entwickler auch eigene Pakete erstellen, die dann von der Applikation verwendet werden können. Da Javascript keine Interfaces unterstützt, wird stattdessen Typescript für die Implementierung verwendet. Damit kann sichergestellt werden, dass Entwickler auf fest definierte Interfaces und Typdefinitionen zurückgreifen können, an die sich ihre Pakete halten müssen.

4.3 Vorgehensmodell

In folgendem Abschnitt wird kurz auf die Vorgehensweise der zwei Befehle `make:schemas` und `make:tests` eingegangen, welche im Prototyp der Applikation umgesetzt werden sollen.

4.3.1 Generierung von JSON Schemas

Zur Generierung der JSON Schemas werden ein *Decorator* und eine *Factory* implementiert:

- **Decorator**: der Decorator iteriert über alle Endpunkte und darin definierte Request- und Response-Bodies, und wandelt, falls vorhanden, die jeweiligen Open-API Schema Definitionen in valide JSON Schemas um. Dafür werden die bereits in Abschnitt 4.1 beschriebenen Unterschiede berücksichtigt. Anschließend werden die erstellten JSON Schemas dem API-Modell hinzugefügt.
- **Factory**: die Factory schreibt die generierten JSON Schemas in `json`-Dateien in den konfigurierten Output-Ordner, und verwendet dabei einfach verständliche Dateinamen. So wird das JSON Schema der Anfrage eines HTTP `PATCH` Aufrufs auf die Ressource `/books` z.B. in `requests/books-update.json` gespeichert.

4.3.2 Generierung von Testcases

Zur Generierung von Testcases werden ebenfalls ein neuer *Decorator* und eine *Factory* implementiert. Der Befehl benutzt außerdem den bereits für JSON Schemas definierten *Decorator*, da in den Tests die Antworten vom Server mithilfe von JSON Schemas validiert werden.

- **Decorator**: der Decorator iteriert über alle Endpunkte und erstellt das in Abbildung 9 definierte Testsuite Modell. Für jede Anfrage werden mithilfe von verschiedenen Heuristiken Parameterwerte abgeleitet, die in den Testanfragen benutzt werden können. Die hierfür verwendeten Heuristiken basieren auf den in Ed-douibi, Izquierdo und Cabot (2018) definierten Heuristiken, wurden jedoch für eine erhöhte Genauigkeit noch angepasst bzw. erweitert und an die aktuelle Syntax der OpenAPI v3.0.2. Spezifikation angepasst:

  1. Der Wert eines Parameters `p` kann abgeleitet werden aus: Beispielwerten (`p.example`, `p.examples`, `p.schema.example`, `p.content.example`, `p.content.examples` oder `p.content.schema.example`), Standardwerten (`p.schema.default`, `p.schema.items.default` wenn `p` vom Typ `array` ist, `p.content.schema.default` oder `p.content.schema.items.default`) oder Enums (`p.schema.enum`, `p.schema.items.enum` wenn `p` vom Typ `array` ist, `p.content.schema.enum` oder `p.content.schema.items.enum`).
  2. Der Wert eines Parameters kann abgeleitet werden, wenn eine Anfrage mit einem automatisch generierten Wert (bspw. ein String wenn der Parameter vom Typ `string` ist) erfolgreich ist.
  3. Der Wert eines Parameters kann abgeleitet werden, wenn für die selbe Ressource in einer der vorhanden Operationen ein Attribut gleichen Namens im Schema der Antwort definiert ist.

  Für fehlerhafte Testcases müssen fehlerhafte Parameter- bzw. Attributwerte abgeleitet werden, was mithilfe verschiedener Regeln basierend auf dem Typ und den im Schema definierten Einschränkungen möglich ist, aufgelistet in den Tabellen 1 und 2.

Tabelle 1: Datentypen

Typ	**Regel**
`object`	Ein Objekt, welches das Schema verletzt
`integer` `number`	Zufälliger String der keine Zahlen enthält
`boolean`	Zufälliger String der nicht `true` oder `false` ist

- **Factory**: die Factory generiert die einzelnen Testcases in einer bestimmten Zielsprache. Dafür werden die in den Decorators angelegten Parameterwerte und JSON Schemas verwendet. Für jede API Operation wird, wenn für alle erfoderlichen Parameter (`p.required = true`) Werte abgeleitet werden konnten, für jede mögliche Parameterkombination ein separates Testcase erstellt, in dem eine Anfrage an den Server mit den abgeleiteten Parameterwerten geschickt wird. Anschließend wird geprüft, dass die Antwort einen validen Statuscode hat (2xx bei nominalen Testcases, bzw. 4xx bei fehlerhaften Testcases), und mit den generierten JSON Schemas validiert werden kann. Über *Hooks*, die bei jedem Testcase vor und nach dem Ausführen der Anfrage aufgerufen werden, kann der Entwickler Anpassungen vornehmen. So kann z.B. Authentifizierung ermöglicht werden.

Tabelle 2: Einschränkungen

Einschränkung	**Regel**
`enum`	String oder Zahl, die nicht im Enum enthalten ist
`maximum / exclusiveMaximum`	Integer, der größer als das Maximum ist
`minimum / eclusiveMinimum`	Integer, der kleiner als das Minimum ist
`minLength`	String, der kürzer als `minLength` ist
`maxLength`	String, der länger als `maxLength` ist
`maxItems`	Array mit mehr Items als `maxItems`
`minItems`	Array mit weniger Items als `minItems`
`format`*`:date-time`* `format`*`:email`* `format`*`:hostname`* `format`*`:ipv4`* `format`*`:ipv6`* `format`*`:uri`*	Zufälliger String aus `[A-Za-z]`

5 Fazit

Mit der Umsetzung dieses Projektes konnte erfolgreich ein Prototyp implementiert werden, der demonstriert, dass auf Basis einer API Spezifikation nominale und fehlerhafte Testcases automatisch generiert werden können. Für die Evaluierung des Prototyps wurde eine Beispielanwendung implementiert, die eine RESTful API für Reisebücher umsetzt. Hierfür wurde das Laravel-Framework benutzt. Bei insgesamt 20 Endpunkten mit verschiedenen Query-Parametern werden 112 Testcases generiert, und damit eine Testabdeckung von 88% erreicht.

Es konnten erfolgreich die operativen Ziele 1, 2, 3, und 6, bzw. die Anforderungen 1-9 umgesetzt werden, was über den initial definierten Funktionsumfang hinausgeht. Dennoch sind noch zahlreiche Ausbaumöglichkeiten vorhanden. Abgesehen von sämtlichen Zielen zu Dokumentationsartefakten, gibt es auch beim Generieren von Testcases noch viele Verbesserungsmöglichkeiten, die in einer späteren Iteration bzw. Fortführung des Projektes umgesetzt werden könnten:

- Partielle Überschreibung von Testcases bei Neugenerierung.
- Unterstützung bei der Erstellung von semantischen Tests.
- Umsetzung weiterer Heuristiken zur Ableitung von Parameter- und Attributwerten.
- Umsetzung weiterer Assertions in den Testcases.
- Möglichkeit, Testcases zu löschen und diese bei Neugenerierung nicht wieder anzulegen (z.B. nötig bei sich gegenseitig ausschließenden Parametern).
- Evaluation des Tools mit Projekten des Unternehmens.
- Abstraktes Datenmodell und Unterstützung mehrerer Spezifikationsformate.

- Generierung von Output in weiteren Sprachen bzw. Frameworks (insbesondere Symfony).

Literatur

Arcuri, Andrea (2017). „RESTful API Automated Test Case Generation“. In: *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*. IEEE, S. 9–20.

Bai, Xiaoying u. a. (2005). „WSDL-based automatic test case generation for web services testing“. In: *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*. IEEE, S. 207–212.

Bartolini, Cesare u. a. (2009). „WS-TAXI: A WSDL-based testing tool for web services“. In: *2009 International Conference on Software Testing Verification and Validation*. IEEE, S. 326–335.

Battle, Robert und Edward Benson (2008). „Bridging the semantic Web and Web 2.0 with representational state transfer (REST)“. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 6.1, S. 61–69.

Chakrabarti, Sujit Kumar und Prashant Kumar (2009). „Test-the-rest: An approach to testing restful web-services“. In: *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:* IEEE, S. 302–308.

Ed-douibi, Hamza, Javier Luis Cánovas Izquierdo und Jordi Cabot (2018). „Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach“. In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, S. 181–190.

Fertig, Tobias und Peter Braun (2015). „Model-driven testing of restful apis“. In: *Proceedings of the 24th International Conference on World Wide Web*. ACM, S. 1497–1502.

Fielding, Roy T und Richard N Taylor (2000). *Architectural styles and the design of network-based software architectures*. Bd. 7. University of California, Irvine Irvine, USA.

Jochen Ludewig und Horst Lichter (2007). *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Dpunkt.Verlag GmbH.

Ma, Chunyan u. a. (2008). „WSDL-based automated test data generation for web service“. In: *Computer Science and Software Engineering, 2008 International Conference on*. Bd. 2. IEEE, S. 731–737.

Martin, Evan, Suranjana Basu und Tao Xie (2006). „Automated robustness testing of web services“. In: *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS 2006)*.

Pezoa, Felipe u. a. (2016). „Foundations of JSON schema“. In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, S. 263–273.

Pinheiro, Pedro Victor Pontes, Andre Takeshi Endo und Adenilso Simao (2013). „Model-based testing of restful web services using uml protocol state machines“. In: *Brazilian Workshop on Systematic and Automated Software Testing*, S. 1–10.

Rodriguez, Carlos u. a. (2016). „REST APIs: a large-scale analysis of compliance with principles and best practices“. In: *International Conference on Web Engineering*. Springer, S. 21–39.

Scherer, Anton (2016). „Description languages for REST APIs-state of the art, comparison, and transformation“. Magisterarb.

Sturgeon, Phil (2018a). *JSON API, OpenAPI and JSON Schema Working in Harmony*. Abgerufen am 28. November 2018. URL: https://blog.apisyouwonthate.com/json-api-openapi-and-json-schema-working-in-harmony-ad4175f4ff84.

— (2018b). *Solving OpenAPI and JSON Schema Divergence*. Abgerufen am 28. November 2018. URL: https://blog.apisyouwonthate.com/openapi-and-json-schema-divergence-part-2-52e282e06a05.

— (2018c). *The Many Amazing Uses of JSON Schema: Client-side Validation*. Abgerufen am 06. November 2018. URL: https://blog.apisyouwonthate.com/the-many-amazing-uses-of-json-schema-client-side-validation-c78a11fbde45.

Tilkov, Stefan u. a. (2015). *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt. verlag.

Xu, Wuzhi, Jeff Offutt und Juan Luo (2005). „Testing web services by XML perturbation“. In: *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 10–pp.