Masterstudiengang Informatik und Kommunikationssysteme

Automatisiertes Testen von RESTful Webservices zur Validierung von Claim-basierten Berechtigungskonzepten mittels der OpenAPI-Dokumentation

Eingereicht von:	Benjamin Kissmann
Geboren am:	16.07.1994
Matrikel-Nr.:	21764
Praxisbetrieb:	Dataport AöR
Hochschulbetreuer:	Prof. Dr. Sven Karol
Betriebliche Betreuer:	Dipl.-Inf. Matthias Schäfer
Abgabetermin:	14.06.2021

ABSTRACT

Diese Masterarbeit hat die Entwicklung eines Test-Tools als Ziel. Dieses Werkzeug soll es ermöglichen die Zuverlässigkeit sowie Claim-basierte Berechtigungskonzepte von RESTful Webservices anhand ihrer OpenAPI-Dokumentation zu überprüfen. Dabei wird untersucht, ob sich das Verhalten des zu testenden Systems mittels zustandsabhängiger Eigenschaften spezifizieren lässt und welchen Einfluss diese auf das Finden von Fehlern haben. Um diese Frage zu beantworten, wurde mit dem entwickelten Test-Tool, REST-QA, ein Teil der GitLab-Schnittstelle validiert. Die Auswertung hat gezeigt, dass die vorgeschlagenen zustandsabhängigen Eigenschaften im generischen Kontext nicht allgemeingültig sind. Mit diesen Eigenschaften konnten in GitLab keinerlei Fehler gefunden werden. Jedoch kann damit keine Aussage über den tatsächlichen Nutzen dieser getroffen werden, da die Validierung nur an einem einzelnen RESTful Webservice durchgeführt wurde. Dessen ungeachtet konnte REST-QA zwei neue Fehler in der GitLab-Schnittstelle entdecken. Die Arbeit stellt somit ein weiteres Beispiel für die Machbarkeit und den Nutzen des zu Grunde liegenden Ansatzes für das Testen von RESTful Webservices mittels der OpenAPI-Dokumentation dar.

ABSTRACT

This master’s thesis targets the development of a testing tool which enables the validation of the reliability and accordance to claim-based authorization schemes of RESTful web services by making use of the OpenAPI specification. This paper introduces stateful properties which specify the behavior of the system under test. The feasibility of these properties and their capability in terms of fault finding are researched. To answer this question the testing tool called REST-QA is used to validate GitLab’s REST interface. The evaluation has shown that stateful properties are not universally valid in a generic context. Moreover, these properties were not capable of finding bugs in GitLab’s interface. As only a single RESTful web services was investigated. the gains of their usage cannot be concluded by this paper. Albeit, REST-QA was able to find two new bugs in GitLab’s interface. Therefore, another example could be shown for the feasibility and benefit of the fundamental approach of using the OpenAPI specification for the testing of RESTful web services.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

.....	.....
Ort, Datum	Unterschrift der Verfasserin/ des Verfassers

INHALTSVERZEICHNIS

1 EINLEITUNG 1

2 THEORETISCHE GRUNDLAGEN 3

2.1 RESTful Webservices 3

2.1.1 REST-Architekturstil.....3

2.1.2 REST und HTTP.....5

2.1.3 Reifegrade von RESTful Webservices.....13

2.2 OpenAPI Specification 14

2.3 Claim-basierte Berechtigungskonzepte 19

2.4 Random Testing 21

2.5 Property-Based Testing 23

3 KONZEPTENTWICKLUNG 26

3.1 Systemkontext 26

3.2 Auswahl der Properties 27

3.3 Analyse der OAS 30

3.4 Aufbau des Test-Tools 38

3.4.1 Aufbau der Testkonfiguration.....40

3.4.2 Aufbau der Test-Runner-Komponente.....44

3.5 Generieren von Testsequenzen und -werten 47

4 VALIDIERUNG 54

4.1 Testaufbau und Durchführung 54

4.2 Ergebnisse 58

4.3 Diskussion 63

5 FAZIT UND AUSBLICK 66

A QUELLEN- UND LITERATURVERZEICHNIS 69

B ABBILDUNGSVERZEICHNIS 74

C LISTING-VERZEICHNIS 75

D TABELLENVERZEICHNIS 76

E ABKÜRZUNGSVERZEICHNIS 77

F ANHANG 78

1 EINLEITUNG

RESTful Webservices sind der De-facto-Standard bei der synchronen Service-zu-Service-Kommunikation (vgl. Schermann et al. 2016). Trotzdem existieren noch keine ausgereiften Werkzeuge, die die Zuverlässigkeit und Sicherheit dieser Schnittstellen automatisiert testen (vgl. Atlidakis et al. 2019). Die meisten der vorhandenen Werkzeuge schneiden dabei den Datenverkehr der Webservices mit, um diesen dann wieder, in veränderter Form, als Anfrage an das *System under Test* (SUT) zu senden (vgl. ebd.). Mit der OpenAPI-Dokumentation existiert eine standardisierte, technische Spezifikation für die Schnittstellen von RESTful Webservices mit einem Marktanteil von 70-80% (vgl. Strnadl 2020). Statt der Wiedergabe des Datenverkehrs kann die OpenAPI-Dokumentation analysiert werden, um so automatisiert Testfälle zu generieren, wobei die Dokumentation selbst als Testorakel herangezogen wird. Anknüpfend an die Arbeiten von Karlsson et al. 2020, Atlidakis et al. 2019 und Viglianisi et al. 2020 soll überprüft werden, ob in diesem Kontext zustandsabhängige Properties genutzt werden können und welchen Einfluss diese auf das Finden von Fehlern im SUT haben. Als Properties werden in dieser Arbeit Eigenschaften bezeichnet, die das Verhalten des SUT spezifizieren. Das Ziel dieser Arbeit ist es, ein Test-Tool zu entwickeln, das in der Lage ist, die Zuverlässigkeit und Claim-basierte Berechtigungskonzepte von RESTful Webservices anhand ihrer OpenAPI-Dokumentation zu überprüfen. Mit Hilfe dieses Test-Tools soll anhand der Schnittstelle der *Community Edition* von GitLab, einer offenen *DevOps*-Plattform, die Anwendbarkeit und der Nutzen von zustandsabhängigen Properties validiert werden (vgl. GitLab o. J. a).

Der Aufbau dieser Arbeit gliedert sich in vier Abschnitte. Zu Beginn werden die relevanten technischen Grundlagen behandelt. Dabei wird auf den REST-Architekturstil und in diesem Rahmen auf das *Hypertext Transfer Protocol* (HTTP) sowie die Reifegrade von RESTful Webservices eingegangen. Darauf folgt der Einblick in die *OpenAPI Specification* (OAS). Die nächste Thematik bilden die Claim-basierten Berechtigungskonzepte. Den Abschluss der theoretischen Grundlagen bilden die Ausführungen zum *Random Testing* und *Property-Based Testing*. Im nächsten Hauptabschnitt wird die Konzeption des Test-Tools thematisiert. Hier wird mit der Beschreibung des Systemkontexts begonnen, gefolgt von der Auswahl der testbaren Properties für RESTful Webservices. Dabei wird auch auf die besagten zustandsabhängigen Properties eingegangen. Im Anschluss werden die Möglichkeiten und Limitationen der automatischen Analyse der OAS diskutiert. Mit dieser Vorarbeit wird darauf der Aufbau des Test-Tools, mit dem Namen *REST-QA*, betrachtet. Im letzten Teil der Konzeption wird die Generation von Testsequenzen und -werten betrachtet. Der anschließende Abschnitt der Validierung beginnt mit der Beschreibung des Testaufbaus und der Durchführung. Als Nächstes werden hier die erzielten Ergebnisse dargestellt und diskutiert. Die Arbeit wird mit einem Fazit und dem Ausblick auf anschließende Forschung abgeschlossen.

2 THEORETISCHE GRUNDLAGEN

In diesem Abschnitt werden die für diese Arbeit wesentlichen technischen Grundlagen und Konzepte erläutert. Zunächst werden die RESTful Webservices betrachtet, die das *System under Test* (SUT) im späteren Verlauf darstellen. Anschließend wird auf die Schnittstellenbeschreibung von RESTful Webservices, konkret die *OpenAPI Specification* (OAS) eingegangen. Darauf folgen die Ausführungen zu den Claim-basierten Berechtigungskonzepten. Im Anschluss werden dann die Testmethode *Random Testing* und *Property-Based Testing* (PBT) erläutert.

2.1 RESTful Webservices

Fielding 2000 hat in seiner Dissertation den *Representational State Transfer* (REST) als einen Softwarearchitekturstil für verteilte, hypermediale Systeme vorgestellt. In anderen Worten hat Fielding damit eine Empfehlung für die Softwarearchitektur des *World Wide Web* (WWW) präsentiert. Das WWW ist seit dem Jahr 2000 stark gewachsen und verdankt sein heutiges Ausmaß dabei genau diesem Architekturstil (vgl. Tilkov et al. 2015). Die Dienste im WWW, die den REST-Architekturstil umsetzen, werden als RESTful Webservices bezeichnet (vgl. Pautasso 2014).

2.1.1 REST-Architekturstil

Im Folgenden wird auf die dem REST-Architekturstil zu Grunde liegenden Konzepte nach Fielding 2000 eingegangen. Die Softwarearchitektur basiert auf dem Client-Server-Modell. Die Kommunikation zwischen den Teilnehmern erfolgt grundsätzlich zustandslos. Das bedeutet, dass alle Informationen, die benötigt werden, um eine

```
{
  "self": "/players/12345",
  "budget": 750.00,
  "links": {
    "deposit": "/players/12345/deposit",
    "limit": "/players/12345/limit",
    "status": "/players/12345/status"
  }
}
```

Listing 2.1: Repräsentation einer Ressource mit Ressourcenidentifikatoren in JSON-Notation

Anfrage zu interpretieren, Teil der Nachricht sein müssen. Um dabei die Anfragelast über das Netzwerk zu reduzieren, sieht REST die Verwendung von Caching vor. Den Kern von REST bilden Ressourcen. Dabei gibt es keine Einschränkung, was als Ressource bezeichnet werden kann. Eine bestimmte Ressource lässt sich über einen Ressourcenidentifikator identifizieren. Ressourcen sind eine Abstraktion und entsprechen der Zugehörigkeitsfunktion $M_R(t)$, die zum Zeitpunkt t einem Ressourcenidentifikator R eine Repräsentation zuordnet (vgl. Fielding 2000, S. 88).

Im Listing 2.1 ist beispielhaft eine Zuordnung zum Ressourcenidentifikator `/players/12345` dargestellt. Das Ergebnis wird hier in der *JavaScript Object Notation* (JSON) repräsentiert. Das JSON-Format ist im *Request for Comments* (RFC) 8259 spezifiziert (vgl. Bray 2017). Die Repräsentation im Listing beinhaltet dabei sowohl die Ressourcenrepräsentationen in Form der Schlüsselwertpaare für `id` und `budget` als auch ein Objekt mit Ressourcenidentifikatoren mit dem Schlüssel `links`. Wenn die gleiche Anfrage für denselben Ressourcenidentifikator zu einem anderen Zeitpunkt wiederholt wird, kann das Ergebnis ein anderes sein, weil beispielsweise das Spielerobjekt bearbeitet wurde. REST fordert nur, dass sich die Bedeutung des Ressourcenidentifikators nicht ändert. Im Beispiel wäre die Bedeutung des Res-

sourcenidentifikators, dass sich die Anfrage auf die aktuelle Version des Spielers mit dem Identifikator `12345` bezieht. Ein weiterer Aspekt von REST ist im Beispiel durch die Ressourcenidentifikatoren im `links`-Objekt zu erkennen. Abhängigkeiten zwischen Ressourcen werden bei REST nach dem Prinzip *hypermedia as the engine of application state* (HATEOAS) realisiert. Konkret bedeutet dies, dass Hyperlinks verwendet werden, um Verbindungen zwischen den Ressourcen abzubilden (vgl. Conklin 1987). Zusammenfassend besteht die Repräsentation einer Ressource also aus einer Menge an Ressourcenrepräsentationen und/oder Ressourcenidentifikatoren. Sämtliche Interaktionen zwischen Anwendungen erfolgen bei REST auf Basis dieser Repräsentationen. Dabei wird entweder der aktuelle oder beabsichtigte Zustand einer Ressource übermittelt.

REST-Anwendungen müssen, im Rahmen der dargestellten technischen Bedingungen, eine definierte Schnittstelle für die Kommunikation anbieten. Dazu zählen die Verwendung von Ressourcenidentifikatoren zum Identifizieren von Ressourcen, die Manipulation der Ressourcen mittels ihrer Repräsentation, die Kommunikation über selbstbeschreibende Nachrichten und die Anwendung von HATEOAS.

2.1.2 REST und HTTP

Wie bereits genannt, wurde der REST-Architekturstil direkt für verteilte, hypermediale Systeme wie das WWW entwickelt. Das *Hypertext Transfer Protocol* (HTTP) wurde ebenfalls gezielt für die Anwendungsschicht solcher Systeme entwickelt (vgl. Fielding et al. 1999). Mittels REST wurden grundlegende Schwächen in den frühen Versionen von HTTP aufgedeckt und behoben (vgl. Fielding 2000). Der Einfluss von REST auf das Protokoll manifestierte sich darüber hinaus im Umfang der standardisierten Erweiterungen. So wurden nur solche

Erweiterungen aufgenommen, die im Einklang zur REST-Architektur stehen (vgl. ebd.). Obwohl der REST-Architekturstil nicht an ein konkretes Protokoll gebunden ist, wird in der Praxis primär das *Hypertext Transfer Protocol* (HTTP) genutzt (vgl. Tilkov et al. 2015). In dieser Arbeit wird REST deshalb ausschließlich im Kontext von HTTP betrachtet. Dieser Abschnitt soll die Spezifika von HTTP im Kontext von REST erörtern.

Das Protokoll ist in der Version HTTP/1.1 im RFC 2616 definiert (vgl. Fielding et al. 1999). Es wird grundsätzlich zwischen Anfrage- und Antwortnachrichten unterschieden. Jede Nachricht besteht aus einer Startzeile, einer beliebigen Anzahl an Header-Feldern und einem optionalen Body. Der Body beinhaltet die *entity*, also die Ressourcenrepräsentation, auf die sich die Nachricht bezieht. Die Header-Felder unterteilen sich in allgemeine Header-Felder, Anfrage-, Antwort- und Entity-Header-Felder. Die Header-Felder ermöglichen es, zusätzliche Information allgemein zur Nachricht, spezifisch für die Anfrage bzw. Antwort oder Metadaten zur Entity zu übermitteln. Darüber hinaus können Header-Felder auch für sog. Cookies verwendet werden (vgl. Fielding 2000). Mittels Cookies können intransparente Daten übermittelt werden. Diese sind in der Regel für die gesamte Webseite gültig und enthalten Information zum Zustand, die unabhängig von der aktuellen Repräsentation sind. Der Server fordert dabei den Client in einer Antwortnachricht auf, die enthaltenen Cookies allen zukünftigen Anfragen beizufügen. Cookies stellen jedoch keine REST-konforme Erweiterung dar, da sie zum einen nicht selbstbeschreibend sind und darüber hinaus gegen die geforderte Zustandslosigkeit der Kommunikation verstoßen. Aus diesem Grund werden Cookies in dieser Arbeit nicht weiter betrachtet.

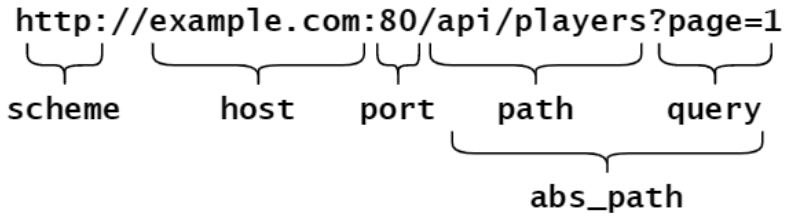
100	Continue	404	Not Found
101	Switching Protocols	405	Method Not Allowed
200	OK	406	Not Acceptable
201	Created	407	Proxy Authentication Required
202	Accepted	408	Request Timeout
203	Non-Authoritative Information	409	Conflict
		410	Gone
204	No Content	411	Length Required
205	Reset Content	412	Precondition Failed
206	Partial Content	413	Request Entity Too Large
300	Multiple Choices	414	Request-URI Too Long
301	Moved Permanently	415	Unsupported Media Type
302	Found	416	Requested Range Not Satisfiable
303	See Other		
304	Not Modified	417	Expectation Failed
305	Use Proxy	500	Internal Server Error
307	Temporary Redirect	501	Not Implemented
400	Bad Request	502	Bad Gateway
401	Unauthorized	503	Service Unavailable
402	Payment Required	504	Gateway Timeout
403	Forbidden	505	HTTP Version Not Supported

Tabelle 2.1: HTTP-Statuscodes (vgl. Fielding et al. 1999, S. 27)

Bevor auf den Aufbau von Anfrage- und Antwortnachrichten eingegangen wird, werden zunächst die HTTP-Statuscodes, die bei HTTP verwendeten Ressourcenidentifikatoren und die HTTP-Methoden betrachtet. HTTP-Statuscodes sind eine Zahlenkombination aus drei Ziffern und codieren das Ergebnis des Versuchs eine Anfrage zu verstehen und zu erfüllen. Die Tabelle 2.1 zeigt die für HTTP/1.1 spezifizierten Statuscodes und die dazu empfohlenen *Reason-Phrases* (vgl. Fielding et al. 1999, S. 27). Während die Statuscodes für die Interpretation durch Maschinen intendiert sind, richten sich die Reason-Phrases an Menschen. HTTP-Statuscodes lassen sich grundsätzlich erweitern und ein Client muss nicht zwingend in der Lage sein, alle Statuscodes zu interpretieren. Stattdessen muss er aber jede Statuscodeklasse interpretieren können. Die Klasse wird jeweils durch die erste Ziffer des dreistelligen Codes bestimmt. Kennt ein Client einen bestimmten Statuscode nicht, so soll die Antwort äquivalent zum `x00`-

Statuscode der Klasse behandelt werden. Kann der Client beispielsweise den Statuscode `422` nicht interpretieren, so muss er stattdessen auf den Statuscode `400` zurückfallen. Statuscodes, die mit einer Eins beginnen, sind informeller Natur und zeigen an, dass die Anfrage erhalten wurde und der Prozess fortgesetzt wird. Beim Statuscode `100` soll der Client mit der Anfrage fortfahren und den verbleibenden Teil der Anfrage senden oder, falls die Anfrage bereits vollständig gesendet wurde, diese Nachricht ignorieren. Die führende Zwei signalisiert das erfolgreiche Empfangen, Verstehen und Akzeptieren der Anfrage. Der Statuscode `200` steht genau für diese Aussage und deutet zusätzlich, je nach Anfrage, auf das Existieren eines Body hin. Die Drei deutet auf eine Weiterleitung hin. In diesem Fall muss der Client weitere Aktionen ausführen, damit die Anfrage abgeschlossen werden kann. Der Statuscode `300` steht für *Multiple Choices*. Die Antwort sollte eine Auswahl an möglichen Ressourcenidentifikatoren und Repräsentationstypen besitzen, aus denen der Client wählen kann. Auf die durch den Server bevorzugte Repräsentation sollte im `Location`-Header-Feld verwiesen werden. Fehler, die auf den Client zurückzuführen sind, beginnen mit einer Vier. Der Statuscode `400` weist darauf hin, dass der Server die Anfrage aufgrund fehlerhafter Syntax nicht interpretieren konnte. Der Client sollte die Anfrage nicht wiederholen. Statuscodes, die auf einen Serverfehler hinweisen, beginnen mit einer Fünf. In diesem Fall war der Server nicht im Stande, eine scheinbar valide Anfrage zu bearbeiten. Mit dem Statuscode `500` wird ausgesagt, dass der Server aufgrund einer nicht erwarteten Bedingung die Anfrage nicht ausführen konnte. Auf Basis dieser Fehlerklassen und `x00`-Statuscodes lassen sich also auch unbekannte Statuscodes interpretieren.

Da sich die Anfragen an den Server, mit wenigen Ausnahmen, immer auf eine Ressource beziehen, soll in diesem Absatz auf die Implementierung der Ressourcenidentifikatoren im Sinne von REST eingegangen werden. HTTP verwendet hierfür *Uniform Resource Identifiers* (URI) (vgl. Fielding et al. 1999). Die aktuelle Version von URI ist im RFC 3986 spezifiziert (vgl. Berners-Lee et. al. 2005). URI lässt sich nicht nur im Kontext von HTTP verwenden, sondern unterstützt auch andere Protokolle wie *ftp*, *ldap* oder *mailto,* um einige Beispiele zu nennen. In dieser Arbeit wird sich aber auf die Verwendung in HTTP beschränkt. URI ist rein zur Identifikation von Ressourcen spezifiziert. Anderen Protokollen ist es aber erlaubt, der URI zusätzliche Semantik zuzuweisen. HTTP erweitert die Spezifikation jedoch nicht. Eine an HTTP adaptierte URI ist dabei wie folgt aufgebaut (vgl. *Berners-Lee et. al. 2005, S. 16):*



Sie beginnt bei HTTP immer mit `scheme` „http". Die `host`- und `port`-Komponenten identifizieren den Server, an den sich die Anfrage richtet. HTTP ist an kein bestimmtes Transport-Protokoll gebunden, in der Regel kommt aber TCP/IP zum Einsatz. Bei der Verwendung von TCP/IP kann es sich beim Host entweder um einen Domänennamen wie im Beispiel oder um eine IP-Adresse handeln. Die Angabe von `port` ist optional und wird standardmäßig mit dem Wert 80 belegt. Die eigentliche Identifikation der Ressource erfolgt durch `abs_path` und lässt sich in `path` und `query` unterteilen. Das `path`-Element besteht aus einer, in der Regel hierarchisch angeordneten, Sequenz von Pfadsegmenten, die durch „/"-Symbole separiert werden. Die

Angabe dieses Elements ist bei HTTP verpflichtend. Der leere Pfad entspricht dem `path` „/". Das `query`-Element ermöglicht die Angabe von nicht hierarchischen Daten zur Ressourcenidentifikation und ist bei HTTP optional.

Nach der Definition von URI werden in den folgenden Absätzen die verfügbaren HTTP-Methoden näher betrachtet. Im Sinne der einheitlichen REST-Schnittstelle existieren für Anfragen definierte Methoden mit gegebener Semantik. Die Methoden können dabei insbesondere sicher bzw. idempotent sein. Sichere Methoden verursachen bei der Ausführung keine Seiteneffekte. Zum Teil werden jedoch trotzdem gezielt Seiteneffekte bei sicheren Methoden implementiert. Fielding et al. 1999 (S. 34) sagt dazu aus: *„The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them.*" Eine idempotente Methode kann wiederholt hintereinander ausgeführt werden und versetzt die Ressource in denselben Zustand wie das einmalige Ausführen der Methode. Sichere Methoden sind auch immer idempotent.

Für HTTP/1.1 sind die Methoden `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE` und `CONNECT` definiert. Im Folgenden werden die vier für diese Arbeit relevanten Methoden näher beschrieben. Die `GET`-Methode ruft genau die Ressource ab, die über die URI der Anfrage identifiziert wird. Die Anfrage kann dabei durch das Setzen eines Header-Felds wie `If-Modified-Since`, `If-Unmodified-Since`, `If-Match`, `If-None-Match` oder `If-Range` zu einem bedingten Abruf geändert werden. Ebenso ändert sich die Bedeutung zu einem partiellen Abruf, wenn das `Range`-Header-Feld gesetzt wurde. Da es sich um eine rein lesende Operation handelt, ist `GET` eine sichere Methode.

Die POST-Methode erlaubt das Hinzufügen einer neuen, untergeordneten Ressource zu der Ressource, die über die URI identifiziert wurde. Die neue Ressource wird im Body der Anfrage übermittelt. Das erfolgreiche Erstellen der Ressource wird mit dem Statuscode 201 (Created) bestätigt. Der Ressourcenidentifikator zur neuen Ressource wird dabei im Location-Header-Feld angegeben. Im Erfolgsfall muss nicht zwangsläufig eine neue Ressource entstehen, die sich mittels einer URI adressieren lässt. Wenn dies der Fall ist, sollte entweder der Statuscode 200 (OK) oder 204 (No Content) zurückgegeben werden.

Die PUT-Methode fordert an, dass die im Body übertragene Ressource an der durch die URI identifizierten Stelle gespeichert werden soll. Falls bereits eine Ressource für die URI besteht, soll die in der Anfrage enthaltene Ressource als modifizierte Version der existierenden behandelt werden. Sollte noch keine Ressource existieren, kann der Server, falls dies möglich ist, die übermittelte Ressource für diese URI neu erstellen. Der Server muss, falls eine neue Ressource angelegt wurde, mit dem Statuscode 201 (Created) antworten. Bei PUT handelt es sich somit um eine idempotente Methode. Der Unterschied zu POST besteht also hauptsächlich darin, dass sich die URI bei PUT direkt auf die Ressource bezieht, die mit der Anfrage übermittelt wird.

Das Löschen der über die URI identifizierte Ressource wird mittels der DELETE-Methode angefragt. Obwohl dem Client nicht garantiert werden kann, dass die Ressource bei einem erfolgreichen Statuscode tatsächlich gelöscht wurde, sollte der Server nur dann so antworten, wenn er zu diesem Zeitpunkt zumindest beabsichtigt, die Ressource zu löschen bzw. unzugänglich zu machen. Im Fall, dass die Aktion noch nicht ausgeführt wurde, sollte der Server mit dem Statuscode 202 (Accepted) antworten. Statuscode 204 (No Content) zeigt hin-

gegen an, dass die Ressourcen bereits gelöscht wurde. Die DELETE-Methode ist idempotent. Wie aus den dargestellten Methoden erkenntlich wurde, lassen sich Ressourcen manipulieren, indem Repräsentationen der Ressource übermittelt werden, genauso, wie es REST fordert.

Nach den Beschreibungen der HTTP-Statuscodes, der URI und den HTTP-Methoden soll nun auf die Anfragenachrichten eingegangen werden. Eine simple Anfragenachricht kann wie folgt aufgebaut sein (vgl. Fielding et al. 1999, S. 25):

```
1 GET /api/players?page=1 HTTP/1.1
2 Host: example.com:80
```

Die Startzeile bei einer Anfrage beginnt mit der HTTP-Methode, gefolgt von der Anfrage-URI und der HTTP-Version. Die Anfrage-URI ist dabei entweder eine vollständige URI, nur `abs_path`, nur der `host` und `port` oder ein „*"-Zeichen. Das Letztere trägt die Bedeutung, dass sich die Anfrage auf keine bestimmte Ressource bezieht. Die Anfrage im Listing besitzt in der zweiten Zeile ein `Host`-Header-Feld. Dieser Header hat dieselbe Semantik wie die Kombination aus `host`

```
1  HTTP/1.1 200 OK
2  Content-Length: 207
3  Content-Type: application/json
4
5  [
6    {
7      "self": "/players/12345",
8      "budget": 750.00,
9      "links": {
10       "deposit": "/players/12345/deposit",
11       "limit": "/players/12345/limit",
12       "status": "/players/12345/status"
13     }
14   }
15 ]
```

Listing 2.2: HTTP-Antwort für GET api/players?page=1 (gekürzt)

und `port` der URI. Eine Ressource wird bei HTTP stets aus der Kombination des `Host`-Header-Felds und der Anfrage-URI bestimmt. Das Listing 2.2 zeigt eine zur vorhergehenden Anfrage passende Antwortnachricht. Die Startzeile einer Antwortnachricht beginnt mit der HTTP-Version, gefolgt von dem Statuscode und der Reason-Phrase. In diesem Fall wurde die Protokollversion 1.1 verwendet und der Statuscode `200` bedeutet, dass die Anfrage erfolgreich durchgeführt wurde. Die Antwort besitzt zwei Header-Felder. Das `Content-Length`-Header-Feld gibt die Länge des Body in Bytes an. Der Medientyp der Ressourcenrepräsentation wird durch das `Content-Type`-Header-Feld angegeben. In diesem Fall also JSON. Ab Zeile fünf beginnt dann der Body mit der Repräsentation der angefragten Ressource.

2.1.3 Reifegrade von RESTful Webservices

Obwohl sich viele Webservices als RESTful bezeichnen, erfolgt dies oftmals nicht in vollständiger Übereinstimmung zum REST-Architekturstil (vgl. Pautasso 2014, S. 2). Aus diesem Grund unterscheidet Leonard Richardson die folgenden vier Reifegrade bei der Umsetzung der Anforderungen (vgl. ebd., S. 4-5):

- Level 0: HTTP wird als Tunnel benutzt und alle Nachrichten werden an eine einzige URI via POST gesendet. Die verschiedenen Funktionen des Webservices können nur mittels des Payload der Nachricht unterschieden werden.
- Level 1: Für jede Ressource gibt es eine dedizierte URI. Welche Funktion für die Ressource ausgeführt werden soll oder auch auf welche Instanz der Ressource sich die Anfrage bezieht, muss aus dem Body der POST-Nachricht extrahiert werden.

- Level 2: Die konkreten Instanzen der Ressourcen lassen sich anhand der angefragten URI bestimmen. Des Weiteren werden die HTTP-Verben entsprechend der spezifizierten Semantik angewendet und damit das uniforme REST-Interface implementiert.
- Level 3: Abhängigkeiten zwischen den Ressourcen werden vollständig mittels HATEOAS repräsentiert und ermöglichen es Clients, den Webservice dynamisch zu erkunden, indem die Hyperlinks weiterverfolgt werden.

2.2 OpenAPI Specification

Die *OpenAPI Specification* (OAS), vorher als *Swagger* bekannt, ist eine technische Spezifikation von Schnittstellen der RESTful Webservices und wird von der *OpenAPI Initiative* betreut (vgl. OpenAPI Initiative 2021). Die Version 3.0.3 wurde am 20. Februar 2020 von Miller et. al 2020 veröffentlicht. OAS beschreibt sich selbst wie folgt:

> The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. (Miller et. al 2020)

Die OAS stellt den De-facto-Standard zur Spezifikation von RESTful Webservices mit einem Marktanteil von 70-80% dar (vgl. Strnadl 2020). Im Folgenden wird auf die für diese Arbeit relevanten Bestandteile der Spezifikation eingegangen. In diesem Kontext getroffene Aussagen beziehen sich, falls nicht anders gekennzeichnet, auf die Spezifikation von Miller et. al 2020.

```yaml
openapi: 3.0.1
info:
  title: Example API
  version: 1.0.0
paths:
  /api/Players:
    get:
      responses:
        200:
          description: Success
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Player'
components:
  schemas:
    Player:
      required:
        - budget
      type: object
      properties:
        id:
          type: integer
          format: int64
        budget:
          type: number
          format: double
      additionalProperties: false
```

Listing 2.3: OpenAPI Specification in YAML-Repräsentation

Die OAS für einen RESTful Webservice ist grundsätzlich ein JSON-Objekt, das beispielsweise in JSON oder YAML repräsentiert werden kann. Im Listing 2.3 ist zur Veranschaulichung die OAS einer simplen Webservice-Schnittstelle dargestellt. In der ersten Zeile wird die Version des verwendeten OAS-Standards angegeben. Das `info`-Objekt in den Zeilen zwei bis vier definiert Metadaten zur beschriebenen Schnittstelle. Das `paths`-Objekt, beginnend ab Zeile fünf, enthält die relativen URIs zu den Ressourcen und die dazugehörigen Operati-

onen. Für die relativen Pfade kann auch *Path Templating* genutzt werden, um so Variablen innerhalb des Pfads zu ermöglichen:

```
/api/Players/{id}
```

Parameter werden mit geschweiften Klammern angegeben, so wie `id` im Beispiel. Zu den relativen Pfaden kann für jede HTTP-Methode maximal eine Operation definiert werden. Im Listing wird nur die `GET`-Methode definiert. Eine Operation beinhaltet die Information, wie eine Anfrage an die Schnittstelle zu formulieren ist, und definiert die zu erwartenden Antwortmöglichkeiten. Die Antworten bestehen aus einem HTTP-Statuscode und einem Antwortobjekt. Die Liste der Antworten muss dabei nicht zwingend alle existierenden Statuscodes beinhalten. Stattdessen sollte aber mindestens eine Antwort für das erfolgreiche Ausführen der Operation sowie Antworten für alle bekannten Fehler definiert sein. Statt der HTTP-Statuscodes können auch Antworten für ganze HTTP-Statuscodeklassen, beispielsweise `2XX`, angegeben werden. Es ist auch möglich, eine Rückfallantwort zu formulieren, für den Fall, dass keine der anderen Statuscodes oder Statuscodeklassen zutreffen. Für die Antworten mit einem HTTP-Body lässt sich der genaue Aufbau im `content`-Objekt beschreiben. Für die Definition des Aufbaus im `schema`-Objekt wird eine Teilmenge aus dem Regelwerk der *JSON Schema Specification* von Wright 2016 mit einigen zusätzlichen Erweiterungen, im folgenden JSON-Schema genannt, verwendet. Es ist hierbei anzumerken, dass im JSON-Schema auch mit Referenzen gearbeitet werden kann, sodass mehrfach verwendete Schemata nur einmal deklariert werden müssen. Das Player-Schema wird im Listing 2.3 oben beispielsweise im `component`-Objekt, konkret ab Zeile 19, definiert und in Zeile 16 über das Schlüsselwort `$ref` referenziert.

```yaml
paths:
  /api/Players/{id}:
    put:
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
            format: int64
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Player'
      responses:
        204:
          description: Success
```

Listing 2.4: Operation mit Parametern und Anfrage-Body in YAML-Repräsentation

Das Listing 2.4 soll noch einmal den Aufbau einer Operation mit Parametern und einem Anfrage-Body verdeutlichen. Hier wird in Zeile vier angegeben, welche Parameter an welcher Stelle der Anfrage benötigt werden. Neben dem `Path` können Parameter auch in Header-Feldern, in der `Query` des Pfads oder auch in Cookies übermittelt werden. Das `requestBody`-Objekt beschreibt den Body, der mit der Anfrage übermittelt werden muss. Das `schema`-Objekt für `parameters` und `requestBody` ist dabei ein JSON-Schema.

Abschließend soll noch auf die Repräsentation von Authentifizierungs- und Autorisierungsinformationen innerhalb der OAS eingegangen werden. Die OAS unterstützt dabei HTTP-Authentifizierung, API-Schlüssel, gewöhnliche OAuth2-Abläufe und *OpenId Connect Discovery*. Die HTTP-Authentifizierung baut auf dem Challenge-Response-Verfahren auf, bei dem der Client nachweisen muss, dass er im Besitz eines *Shared Secret* ist (vgl. Fielding und Reschke 2014).

```yaml
components:
  securitySchemes:
    player_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: http://example.com/api/oauth/dialog
          scopes:
            write:players: modify players
            read:players: read players
```

Listing 2.5: Definition Sicherheitsschemaobjekt für OAuth2-Schema

```yaml
paths:
  /api/Players:
    get:
      responses: {...}
      security:
      - player_auth:
        - read:players
```

Listing 2.6: Verwendung eines Sicherheitsschemas mit OAuth2-Schema

Bei der Authentifizierung mithilfe eines API-Schlüssel muss der Schlüssel bei jeder Anfrage mitgesendet werden. Auf die beiden zuletzt genannten Mechanismen wird ausführlich im Abschnitt 2.3 eingegangen. Innerhalb des `component`-Objekts der OAS müssen die verwendbaren Authentifizierungsschemas deklariert werden. Im Listing 2.5 ist die Definition für ein OAuth2-Schema zur Veranschaulichung dargestellt. Das Sicherheitsschema muss dabei zunächst im `components`-Objekt definiert werden. Anschließend kann das Schema wie im Listing 2.6 für eine bestimmte Operation verwendet werden. Bei OAuth2 oder auch OpenId Connect lassen sich die Sicherheitsanforderungen nochmals feingranularer in einzelne *Scopes* zerlegen. Im Listing wird für das Auslesen der Spieler beispielsweise nur eine Autorisierung für den Scope `read:players` benötigt. Neben der Definition der Sicherheitsvoraussetzungen für jede einzelne Operation lassen sich die Anforderungen auch auf unterster Ebene für die gesamte Schnittstelle deklarieren.

2.3 Claim-basierte Berechtigungskonzepte

Chappell 2011 beschreibt die Claim-basierte Identifizierung als vereinheitlichten Weg für Softwareanwendungen, Zugriff auf Identitätsinformationen eines Nutzers der eigenen Organisation, einer anderen Organisation oder aus dem Internet zu erlangen, und dies unabhängig davon, ob die Software vor Ort oder in der Cloud betrieben wird. Ein *Claim* spiegelt dabei eine bestimmte Information eines Nutzers wider, wie beispielsweise das Alter oder den Namen. Gleichzeitig kann ein Claim aber auch bestimmte Berechtigungen oder Einschränkungen für den Nutzer ausdrücken. Ein *Token* entspricht in diesem Kontext einer Menge aus mindestens einem Claim. Die Ausstellung eines Tokens erfolgt dabei von einem *Identity Provider*. Um die Integrität des Tokens sicherzustellen, wird dieser mit einer digitalen Signatur versehen. Die Abbildung 2.1 stellt den Prozess von der Ausstellung und der Nutzung eines solchen Tokens dar. Im ersten Schritt authentifiziert sich der Nutzer gegenüber dem Identity Provider. Dieser erstellt bei Erfolg einen Token und gibt diesen an den Client zurück. Der Client kann den Token nun im zweiten Schritt nutzen, um

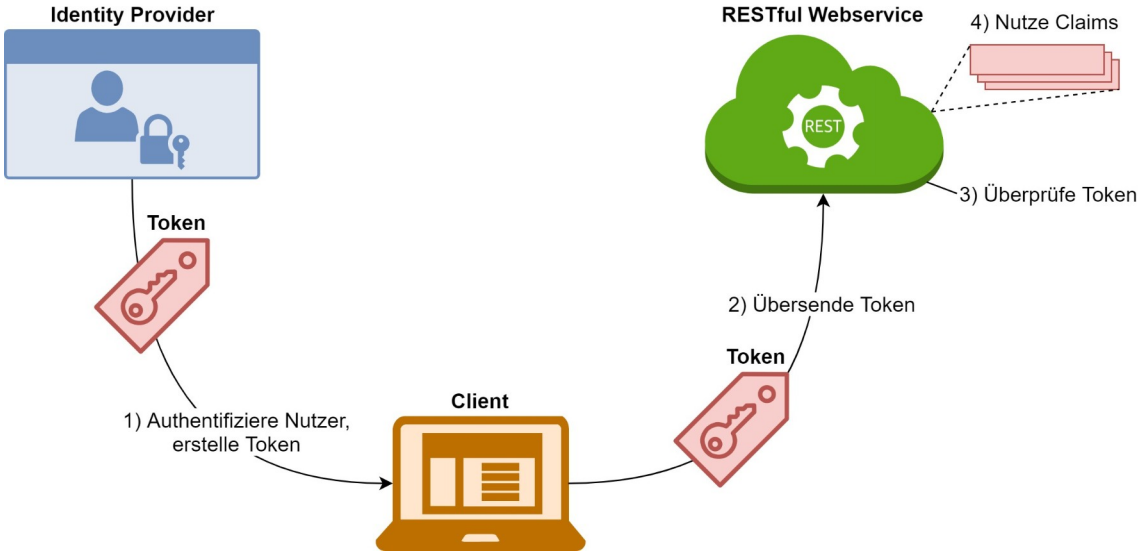


Abbildung 2.1: Ausstellung und Nutzung eines Tokens (nach Chappell 2011, S. 6)

sich gegenüber dem RESTful Webservice zu authentifizieren. Der RESTful Webservice überprüft nun den Token. Dabei wird u. a. die Signatur validiert und überprüft, ob der Identity Provider, der den Token ausgestellt hat, vertrauenswürdig ist. Im letzten Schritt kann der RESTful Webservice die im Token enthalten Claims nutzen, um beispielsweise die Berechtigungen des Nutzers zu überprüfen. Berechtigungskonzepte, die auf diesem Mechanismus aufbauen, werden in dieser Arbeit als Claim-basierte Berechtigungskonzepte bezeichnet.

Das *OAuth 2.0 Authorization Framework* stellt einen Rahmen für die Implementierung des eben dargestellten Autorisierungsmechanismus dar (vgl. Hardt 2012). Insbesondere sind hier die Scopes herauszustellen, die den Zugriff auf bestimmte Ressourcen eines RESTful Webservices steuern können. Wie bereits im Abschnitt 2.2 beschrieben, werden die Scopes dazu genutzt, Berechtigungsvoraussetzungen für die Operationen festzulegen. Dabei handelt es sich bei den Scopes innerhalb eines Token um Namen, die vom Tokenausteller vorgegeben werden (vgl. Jones und Hardt 2012). Ein Äquivalent zu den Scopes bilden *Realms*, wie sie bei der HTTP-Authentifizierung genutzt werden (vgl. Fielding und Reschke 2014). Auch sie ermöglichen die Gruppierung der Ressourcen eines Webservices in mehrere Berechtigungsbereiche. *OpenID Connect 1.0* ist eine zusätzliche Identitätsschicht, die auf OAuth 2.0 aufbaut (vgl. Sukimura et al. 2014). OpenID Connect definiert beispielsweise Standard-Claims, wie den Familiennamen oder das Geschlecht für Nutzer, und beschreibt die Schnittstelle zum Auslesen von Benutzer-Claims.

2.4 Random Testing

Bevor auf das eigentliche *Random Testing* eingegangen wird, sollen zunächst einige Begriffe und ihre Bedeutung in dieser Arbeit definiert werden. Parameter bzw. Eingabeparameter bezeichnen die Variablen, mit denen das SUT aufgerufen wird. Veranschaulicht sind `a` und `b` die Eingabeparameter der folgenden Methodensignatur:

```
int Mul(int a, int b)
```

Ein Wert bezeichnet eine konkrete Instanziierung eines solchen Parameters. Für das Beispiel wären potentielle Werte z. B. 31 für `a` und -15 für `b`. In Anlehnung an Fink und Bishop 1997 bezeichnet ein Testfall eine Belegung aller Parameter mit jeweils einem bestimmten Wert. Im Falle von zustandsabhängigen Tests beinhalten Testfälle zusätzlich die Sequenz der Aufrufe an das SUT. Ein Testfall schlägt fehl, wenn bei der Durchführung ein Fehler festgestellt wurde, und wird als bestanden gewertet, wenn dies nicht der Fall ist. Ein Test beschreibt eine Menge an Testfällen und gilt als bestanden, wenn keiner der Testfälle fehlschlägt. Ob ein Testfall fehlerbehaftet ist, entscheidet das sog. Testorakel (vgl. Claessen und Hughes 2000).

Beim Random Testing handelt es sich um eine bestimmte Methode des Black-Box-Testings (vgl. Duran und Ntafos 1984). Im Gegensatz zu systematischen Tests, bei denen die Werte der Parameter systematisch festlegt werden, werden hier laut Hamlet 2002 die Werte pseudozufällig generiert. Die generierten Werte sind dabei im Sinne der Spezifikation des SUT, dem sog. Betriebsprofil, grundsätzlich valide. Es werden also keine inkompatiblen Eingaben produziert. Das Betriebsprofil beschreibt zusätzlich die stochastische Verteilung der zu generierenden Werte. Das Random Testing zielt nicht darauf ab, mit einem Test einen bestimmten Anteil des Quelltext oder der Pro-

grammpfade abzudecken, wie es bei systematischen Tests üblich ist (vgl. ebd.). Hamlet 2002 (S. 13) argumentiert, dass die Wahl einer bestimmten Abdeckung ungeeignet ist: „Yet there is no information on the effectiveness of 80% coverage at finding failures, nor on the significance of an 80%-coverage test that happens to succeed." Dies liegt u. a. in der Tatsache begründet, dass ein von einem Test abgedeckter Programmpfad zwar für die im Test verwendeten Werte eine korrekte Ausgabe erzeugt, aber andere Werte zu einem falschen Resultat führen können (vgl. Howden 1980). Laut Duran und Ntafos 1984 konvergiert die Fehlerrate beim Random Testing gegen einen festen Wert θ, wenn hinreichend viele Testfälle mit dem Betriebsprofil des SUT ausgeführt werden. Im Fall, dass kein Fehler gefunden wird, lässt sich ein Wert θ^* mit der folgenden Formel berechnen (Duran und Ntafos 1984, S. 11):

$$\theta^* = 1-\alpha^{1/n}$$

Mit dieser Formel lässt sich die maximale Fehlerrate in Abhängigkeit von der Anzahl n der Testfälle mit einer bestimmten Irrtumswahrscheinlichkeit α berechnen. Die berechnete maximale Fehlerrate gilt aber immer nur für das genutzte Betriebsprofil. Die Definition eines Betriebsprofils, das die Einsatzumgebung des SUT exakt abbildet, stellt in der Praxis eine große Herausforderung dar (vgl. Tsoukalas et al. 1993). Mit der Ungenauigkeit des Betriebsprofils ist jedoch auch der durch die Formel berechnete Wert beliebig ungenau und erlaubt somit in der Praxis ebenfalls keine zuverlässige Aussage über die Signifikanz des ausgeführten Tests (vgl. Hamlet 2002). Stattdessen eignet sich diese Berechnung als Maß zum Vergleich von verschiedenen Testmethoden innerhalb der Forschung (vgl. Hamlet 2002, S. 12). Obwohl das Random Testing kein bestimmtes Abdeckungsziel verfolgt, haben Duran und Ntafos 1984 festgestellt, dass trotzdem eine sehr hohe Pfadabdeckung erreicht wird. Hamlet 2002 (S. 14)

bestätigt diese Aussage: *„Roughly, by taking 20% more points in a random test, any advantage a partition test might have enjoyed is wiped out.*" Bei den durch das Random Testing nicht abgedeckten Programmpfaden handelt es sich laut Duran und Ntafos 1984 regelmäßig um die Behandlung von Ausnahmefällen. Deshalb schlagen sie ein erweitertes Testen mit Spezial- bzw. Extremwerten vor.

Eine Stärke des Random Testing ist es, sehr einfach eine große Menge an Testfällen zu generieren (vgl. Hamlet 2002). Deshalb ist es essentiell, dass die Validierung der Ausgaben automatisiert erfolgt (vgl. Duran und Ntafos 1984). Damit kommt auch die nicht triviale Frage des Testorakels auf. In der Regel werden für Testfälle Werte gewählt, die eine (für Menschen) nachvollziehbare Klassifikation der Ausgabe erlauben (vgl. ebd.). Dies ist beim Random Testing aber explizit nicht der Fall (vgl. ebd.). Im folgenden Abschnitt wird deshalb die für diese Arbeit relevante Umsetzung des Testorakels erläutert.

2.5 Property-Based Testing

Die Grundidee des *Property-Based Testing* (PBT) ist die Verwendung einer formalen Softwarespezifikation bzw. -dokumentation als Testorakel (vgl. Richardson et al. 1992). Ein Testfall schlägt fehl, wenn das SUT von einer der spezifizierten Eigenschaften, im Folgenden zur klaren Abgrenzung als *Properties* bezeichnet, abweicht (vgl. Fink und Bishop 1997). Bei den Properties kann es sich dabei sowohl um konkrete Ein- und Ausgabepaare handeln als auch um allgemein gehaltene logische Ausdrücke (vgl. Richardson et al. 1992). Zur Verdeutlichung folgt ein Beispiel für eine Property mit definierter Eingabe und Ausgabe. So muss die Funktion MUL bei den Eingabewerten zwei und drei den Wert sechs zurückgeben:

$$MUL(2, 3) = 6$$

Genauso lässt sich auch ein allgemeiner logischer Ausdruck formulieren, der für jede beliebige reelle Zahl x gilt:

$$\forall x \in \mathbb{R}: \; MUL(x, x) \geq 0$$

Zusätzlich zu den Properties aus der Spezifikation existieren auch generische Properties, beispielsweise für Array-Grenzen oder Race Conditions (vgl. Fink und Bishop 1997). Das PBT baut auf der Annahme auf, dass die Gesamtheit der Properties alle relevanten Eigenschaften des SUT beschreibt und damit dieses validieren kann (vgl. ebd.). Da die Eigenschaften aus der formalen Spezifikation abgeleitet werden, wird statt der Korrektheit des Programms vielmehr die Übereinstimmung des SUT mit dieser überprüft (vgl. Richardson et al. 1992). Diese Abhängigkeit bringt dabei mehrere Vorteile mit sich. So ist PBT in der Lage, Fehler in der Spezifikation zu finden (vgl. Claessen und Hughes 2000). Da die Spezifikation das Verständnis der Entwickelnden des SUT widerspiegelt, lassen sich so falsche Annahmen über die Funktionsweise erkennen (vgl. ebd.). Des Weiteren wird die Aktualität der Spezifikation durch die Tests sichergestellt (Peters und Parnas 1994). Beim PBT bekommt damit die Spezifikation des SUT einen essentiellen Stellenwert (vgl. ebd.). Laut Claessen und Hughes 2000 motiviert PBT damit die Entwickelnden, die Spezifikation aktuell zu halten, und verbessert zugleich das Verständnis über das SUT.

Um nun mit Hilfe des PBT das SUT zu überprüfen, wird das Programm mit einer Reihe von Testfällen mit jeweils verschiedenen Eingabewerten ausgeführt (vgl. Fink und Bishop 1997). Dabei ist es grundsätzlich offen, wie diese Testfälle generiert werden (vgl. ebd.). Fink und Bishop 1997 fordern, dass sich die Abdeckung der Tests strukturiert auf Quelltextebene messen lässt, um so die Vollständigkeit nach dem Grad der Abdeckung bestimmen zu können. Solange das ange-

strebte Maß nicht erreicht ist, sollen neue Testfälle generiert werden, die den noch nicht abgedeckten Quelltext gezielt testen (vgl. ebd.). Da es dafür sowohl der Kenntnis des Quelltexts als auch der Instrumentierung des SUT bedarf, handelt es sich bei diesem Vorgehen um einen White-Box-Ansatz. Konträr dazu halten Claessen und Hughes 2000 eine Messbarkeit der Vollständigkeit zwar für nützlich, aber nicht erforderlich. Um Testfälle zu generieren, nutzen sie die Methodik des Random Testing. Im Abschnitt 2.4 wurde bereits erläutert, dass sich damit eine hohe Pfadabdeckung des SUT erreichen lässt, ohne diese gezielt anzustreben. Der Vorteil des Ansatzes von Claessen und Hughes 2000 ist dabei, dass die Generierung von Eingabeparametern nicht vom Quelltext, sondern allein von der Spezifikation abhängig ist und es sich somit um einen Black-Box-Test handelt.

Im Folgenden soll kurz das *Modelbased Testing* betrachtet werden, das viele Gemeinsamkeiten zum PBT aufweist. Utting und Legeard 2010 beschreiben u. a. das Generieren von Eingabewerten für die Testfälle anhand der Spezifikation als eine Disziplin des Modelbased Testing. Dies entspricht dem Ansatz von Claessen und Hughes 2000, da auch beim Random Testing die Testwerte von dem Betriebsprofil, also der Spezifikation, abgeleitet werden. Eine weitere Disziplin ist das Generieren von Testfällen und der entsprechenden Testorakel anhand eines Verhaltensmodel des SUT (vgl. Utting und Legeard 2010). Dies ist vergleichbar mit der Ableitung der Properties von der formalen Dokumentation des SUT. Das PBT kann also auch als eine Spezialform des Modelbased Testing angesehen werden.

3 KONZEPTENTWICKLUNG

In diesem Abschnitt wird nun die Konzeptentwicklung für die Umsetzung von REST-QA betrachtet. Dafür wird zunächst der Kontext der Problemstellung dargestellt. Im Anschluss werden die überprüfbaren Properties und daraufhin die Analyse der OpenAPI Specification (OAS) beschrieben. Darauf folgt die ausführliche Betrachtung des Aufbaus von REST-QA. Im letzten Abschnitt wird dann auf die Generierung von Testfällen eingegangen.

3.1 Systemkontext

Um ein besseres Verständnis für die Problemstellung zu schaffen, ist in Abbildung 3.1 REST-QA im Kontext der umgebenden Systeme dargestellt. Das zu entwickelnde Test-Tool soll dabei zum einen manuell von einer Person genutzt und zum anderen in die Build-Automatisierung des SUT eingebunden werden. REST-QA hat dabei die Aufgabe, die Zuverlässigkeit des SUT sowie die Einhaltung der spezifizierten Berechtigung zu überprüfen. Dabei wird grundsätzlich ein Black-Box-Ansatz verfolgt. Das OAS-Dokument enthält die Schnittstellenspezifikation vom SUT und muss von REST-QA analysiert werden, um auto-

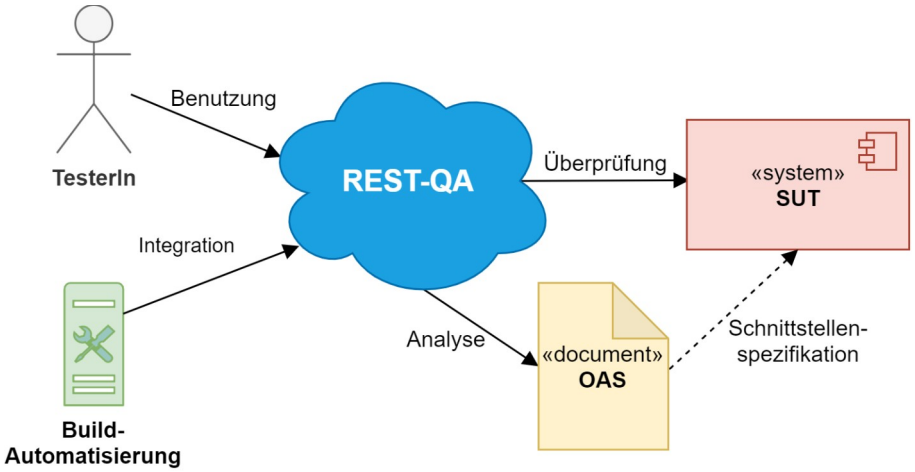


Abbildung 3.1: Kontext der Systemumgebung

matisiert Testfälle zu generieren und auszuführen. Bereits am Kontext des Test-Tools lässt sich erkennen, dass das Testorakel hauptsächlich auf dem OAS-Dokument basieren muss, da es allein die Schnittstelle des SUT spezifiziert. Darüber hinaus handelt es sich beim SUT um einen generischen REST-Webservice. REST-QA ist damit in der Lage, die Übereinstimmung der Implementierung des SUT zum HTTP-Standard und REST-Architekturstil sowie die Übereinstimmung zum OAS-Dokument zu überprüfen. Eine Voraussetzung für den dargestellten Testansatz ist also eine korrekte und aktuelle Dokumentation des SUT mittels des OAS-Dokuments.

3.2 Auswahl der Properties

Nach der Betrachtung des Kontexts der umgebenden Systeme soll nun erörtert werden, welche Properties sich für das SUT automatisiert überprüfen lassen. Wie bereits beschrieben, handelt es sich beim SUT um einen generischen RESTful Webservice. Somit können nur Properties verwendet werden, die in der OAS maschinenlesbar beschrieben werden oder die sich aus dem HTTP-Standard bzw. REST-Architekturstil ableiten lassen. Es lassen sich also keine spezifischen Anforderungen an das SUT testen, die über diese generische Sicht hinausgehen. Bei der Definition eines Modells für das SUT durch die Properties soll laut Hughes 2013 das Modell simpel und verständlich gehalten werden. Er argumentiert weiter, dass beim Property-based Testing die meisten Fehler auch mit stark vereinfachten Modellen gefunden werden können. So gelang es beispielsweise Atlidakis et al. 2019, eine Vielzahl an Fehlern mit nur einer Property – das SUT darf nicht mit dem Statuscode `500` (Internal Server Error) antworten – ausfindig zu machen. Auf Basis dieser Aussage kann also das Modell eines RESTful Webservices auf allgemeingültige Properties reduziert

werden. Trotzdem sollte REST-QA in der Lage sein, eine Vielzahl der Fehler im SUT zu finden.

Grundsätzlich lässt sich im Kontext von RESTful Webservices zwischen zustandslosen und zustandsabhängigen Properties unterscheiden. Zustandslose Properties sind immer unabhängig vom Zustand einer Ressource gültig. Umgekehrt sind zustandsabhängige Properties nur gültig, wenn sich die Ressource in einem bestimmten Zustand befindet. Zunächst sollen die zustandslosen Properties untersucht werden, die für ein SUT in Betracht kommen. Ein primäres Ziel von REST-QA ist die Validierung der Übereinstimmung der Dokumentation mit der Implementierung.

- (P1): Die tatsächliche Antwortnachricht vom SUT muss deshalb stets in der OAS beschrieben werden (vgl. Viglianisi et al. 2020). Dabei muss der Statuscode der Antwort dokumentiert sein, der Aufbau der Ressource im Antwort-Body muss dem JSON-Schema entsprechen und alle zusätzlich enthaltenen Header-Felder müssen ebenfalls dokumentiert sein.
- (P2): Um das Berechtigungskonzept zu validieren, müssen bei einer Anfrage immer alle Sicherheitsanforderungen gewährleistet sein.
- (P3): Umgekehrt muss das SUT mit dem Statuscode `403` (Forbidden) antworten, wenn dies nicht der Fall ist.
- (P4): Ein weiterer Aspekt ist, dass das SUT in der Lage sein muss, jede Anfrage zu bearbeiten, ohne dabei in einen undefinierten Zustand zu geraten. So definieren auch Atlidakis et al. 2019 den Statuscode `500` (Internal Server Error) als Fehler des SUT. REST-QA klassifiziert alle Statuscodes der Klasse `5XX`, also

solche, die einen Fehler des Servers indizieren, obwohl die Anfrage selbst scheinbar korrekt ist, als Fehler des SUT.

- (P5): Darüber hinaus darf bei korrekten Anfragen auch kein Statuscode erzeugt werden, der auf eine falsch konstruierte Anfragenachricht hindeutet, wie z. B. Statuscode `400` (Bad Request). Diese Property lässt sich jedoch aus der Black-Box-Perspektive nicht eindeutig entscheiden und kann deshalb in REST-QA deaktiviert werden. (vgl. Viglianisi et al. 2020).

Für zustandsabhängige Properties lassen sich drei generische Zustände einer Ressource ausmachen. Alle Ressourcen, zu denen das Test-Tool einen Ressourcenidentifikator gespeichert hat, gelten als existierend. Ressourcen, die erfolgreich entfernt wurden, gelten als gelöscht, und Ressourcen, die nie ausgelesen oder erstellt wurden, gelten als unbekannt. Bei den folgenden Properties gilt immer die Voraussetzung, dass die Anfrage grundsätzlich valide ist. Im Falle einer Weiterleitung gelten die Properties für die durch die Weiterleitung referenzierten Ressourcen.

- (P6): Beim direkten Zugriff auf eine unbekannte Ressource muss das SUT mit dem Statuscode `404` (Not Found) antworten. Eine Ausnahme bildet die `PUT`-Methode. Bei dieser Methode kann zusätzlich der Statuscode `201` (Created) vorkommen, falls die Implementierung das Angelegen der Ressource in diesem Fall unterstützt.
- (P7): Beim direkten Zugriff auf eine gelöschte Ressource muss mit dem Statuscode `404` (Not Found) oder Statuscode `410` (Gone) geantwortet werden. Eine Ausnahme bildet wieder die `PUT`-Methode.

- (P8): Beim direkten Zugriff auf eine existierende Ressource wird ein Statuscode der Klasse 2XX erwartet. Ressourcenidentifikatoren werden nicht nur beim direkten Zugriff, sondern auch als Referenz verwendet.
- (P9): Bei der Nutzung einer unbekannten oder gelöschten Ressource als Referenz ist ein Statuscode der Klasse 4XX zu erwarten. In diesem Fall wird also P5 überschrieben.

3.3 Analyse der OAS

Damit sich die genannten Properties überprüfen lassen, muss REST-QA zum einem im Stande sein, gültige Anfragen an das SUT zu senden. Zum anderen muss das Test-Tool das SUT bzw. eine Ressource in einen bestimmten Zustand versetzen können. Da die Interna des SUT beim gewählten Testansatz verborgen bleiben, muss die OAS, die die Schnittstelle des SUT spezifiziert, analysiert werden. Für das automatisierte Testen ist die Kenntnis über die vorhandenen Ressourcen, die zugehörigen Operationen mit ihren Parametern und die Abhängigkeiten zwischen den Ressourcen und Parametern elementar.

Der Aufbau der OAS wurde bereits im Abschnitt 2.2 beschrieben. Die Analyse der OAS schafft die Grundlagen für das anschließende Testen des SUT. Dafür müssen zunächst die Operationen der OAS in durch REST-QA ausführbare Operationen überführt werden. Konkret bedeutet dies die Anwendung der Vererbungs- bzw. Überschreibungsregeln der Parameter, Serverlisten und Sicherheitsanforderungen innerhalb der Hierarchie der OAS. Für die Generierung der Werte zu den Parametern wird jeweils das spezifizierte JSON-Schema genutzt. Mit diesen Informationen ist REST-QA bereits in der Lage, Anfragen mit validen Parametern an das SUT zu senden und die im Abschnitt

3.2 als zustandslos beschriebenen Properties zu überprüfen. Dieser triviale Ansatz scheitert jedoch daran, möglichst alle dokumentierten Statuscodes abzudecken (vgl. Atlidakis et al. 2019). In der Praxis konnte schon für die `GET`-Methode des dargestellten Fallbeispiels der Statuscode `200` (OK) nicht erzeugt werden:

```
POST /api/Players
GET  /api/Players/{id}
```

Im Beispiel handelt es sich bei der `id` um eine Zahl vom Datentyp `Int64`. Die Eintrittswahrscheinlichkeit, zweimal die gleiche `id` zu generieren, ist zu gering, um das SUT selbst in diesem minimalen Fallbeispiel mit einem verhältnismäßigen Aufwand zu testen. Die Analyse der OAS muss demnach um Auswertung der Abhängigkeiten zwischen den Ressourcen und den Parametern erweitert werden.

Wie im Abschnitt 2.1 bereits beschrieben wurde, beziehen sich die Constraints einer REST-Schnittstelle vordergründig auf die angewandte Methodik und schränken den konkreten Aufbau von Ressourcen und Ressourcenidentifikatoren nicht weiter ein. Stattdessen liegt die Verantwortung und die Gestaltungsfreiheit hierfür bei den Entwickelnden der Schnittstelle. Allein an der OAS lässt sich deshalb automatisiert keine zuverlässige Erkennung der Abhängigkeiten zwischen den Ressourcen realisieren. Da diese Arbeit darauf abzielt, das Testen möglichst ohne zusätzlichen Aufwand durch die Testenden zu realisieren, werden im Folgenden Ansätze für die automatische Analyse der OAS vorgestellt und Einschränkungen für das Schnittstellendesign genannt, mit welchen diese erfolgreich durchgeführt werden können.

Das zu Beginn dargestellte Problem der Erstellung eines Spielers und dem anschließenden Auslesen lässt sich bereits mit der Analyse der Pfade lösen. Der REST-Architekturstil spezifiziert, dass sich Res-

sourcen eindeutig durch den Ressourcenidentifikator unterscheiden lassen. Für URI entspricht der Ressourcenidentifikator der Kombination aus `path` und `query`. Für REST-QA wird diese Definition leicht abgewandelt. Da Query-Parameter optional angegeben werden können, werden nur solche als Teil des Ressourcenidentifikators interpretiert, die verpflichtend angegeben werden müssen. Des Weiteren wird nach den Beobachtungen des Autors der Ressourcenidentifikator oftmals vollständig durch `path` ausgedrückt, wie es beispielsweise bei der Schnittstelle von GitLab der Fall ist (vgl. GitLab o. J. c). Aus diesem Grund können Query-Parameter optional bei der Auswertung des Ressourcenidentifikators von REST-QA ausgeschlossen werden. Operationen mit demselben Ressourcenidentifikator beziehen sich also unabhängig von der HTTP-Methode immer auf dieselbe Ressource. Im ersten Schritt der Pfadanalyse kann deshalb versucht werden, zu jedem Pfad die entsprechende Ressourcenklasse abzuleiten. Dafür können die folgenden drei Ansätze genutzt werden. Im einfachsten Fall besitzt der Pfad eine `GET`-Operation. Da die `GET`-Methode exakt die Ressource zurückgibt, die durch die URI identifiziert wird, entspricht der Aufbau der Ressourcenklasse direkt dem JSON-Schema der Antwort, die dem Statuscode `200` (OK) zugeordnet ist. Analog kann bei einer `PUT`-Operation der Aufbau der Ressource in der Anfrage betrachtet werden, denn mit der `PUT`-Methode wird angefordert, dass die übertragene Ressource an der durch die URI identifizierten Stelle gespeichert werden soll. Aus `POST`-Operationen kann ebenfalls eine Annahme getroffen werden. Bei der `POST`-Methode wird zur durch die URI referenzierten Ressource eine untergeordnete Ressource angelegt. Es lässt sich somit die Annahme treffen, dass es sich bei der referenzierten Ressource um eine Liste der Klasse handelt, bei der der Aufbau eines einzelnen Elements in der Anfrage charakterisiert wird. Da weder REST noch HTTP fordern, dass es sich bei der

übergeordneten Ressource um eine Liste handeln muss, sollten stets die beiden zuerst genannten Regeln für die Analyse der Ressourcenklasse bevorzugt werden. Als Voraussetzung muss hierbei der RESTful Webservice mindestens den Reifegrad *Level 2* (vgl. Abschnitt 2.1.3) besitzen.

Aus dem Aufbau der Ressourcenklassen können nun, solange es sich um Objekte und keine primitiven Datentypen handelt, die Abhängigkeiten zwischen den Ressourcen analysiert werden. Hierfür wird der Aufbau der JSON-Schemata miteinander verglichen. Die Voraussetzung ist hierbei, dass jeder Ressourcenklasse exakt ein JSON-Schema zugeordnet wird. Besonders hilfreich ist es dann, wenn die Definition des Aufbaus durch eine Referenz erfolgt. In diesem Fall reicht der Vergleich der Referenzen aus, um die Abhängigkeiten eindeutig zu erkennen. Bei der erfolgreichen Analyse von Listen-Ressourcen werden so auch Eltern-Kind-Beziehungen sichtbar. Beispielsweise für die folgenden zwei Pfade:

```
/api/players        -> Player[]
/api/players/{id}   -> Player
```

Während sich `/api/players` auf eine Liste von Spielern bezieht, identifiziert `/api/players/{id}` ein einzelnes Element dieser Liste. Ein weiterer Indikator ist der Aufbau der beiden Pfade. Die Segmente von `path` einer URI sind in der Regel hierarchisch angeordnet. Damit ist `players` der gesamten API und `id` dementsprechend der `players`-Ressource untergeordnet. Daraus lässt sich schlussfolgern, dass die Ressourcenklasse des ersten Pfads der des zweiten Pfads übergeordnet ist. Im nächsten Schritt können jetzt die Beziehungen der Pfad-Parameter zu den Ressourcen hergestellt werden. Da der `id`-Parameter das letzte Glied in der hierarchischen Kette von Pfadsegmenten ist, ist es genau die Information, die die referenzierte Res-

source von den anderen Ressourcen derselben Klasse unterscheidet. Damit lässt sich `id` also als primärer Identifikator der Ressourcenklasse `player` betrachten. Die so gewonnene Erkenntnis lässt sich analog auf alle Pfade übertragen, die ebenfalls mit dem Teilpfad `/api/players/{...}` beginnen, wie:

```
/api/players/{id}/status
/api/players/{playerId}/deposits/{depositId}
```

In beiden Fällen ist die Bedeutung des ersten Parameters im Pfad dieselbe, unabhängig von der konkreten Benennung des Parameters. Allein der Aufbau des Pfads ist für die Semantik entscheidend. Mit dieser Methodik lässt sich die Bedeutung der einzelnen Parameter des Pfads schrittweise aufschlüsseln. Im Fall, dass kein Pfad existiert, um die Bedeutung mit der dargestellten Vorgehensweise festzustellen, kann versucht werden, den Parameternamen mit den Namen der in der OAS enthaltenen Schemata zu vergleichen. Die Voraussetzungen für die erfolgreiche Analyse sind somit also, dass entweder zu jedem Pfad-Parameter ein Teilpfad existiert, von dem sich die Zuordnung zu einer Ressource ableiten lässt, oder dass sich über den Namen des Parameters diese Verbindung herstellen lässt. Mit der zusätzlichen Information zur Bedeutung der Parameter innerhalb eines Pfads können für Anfragen gezielt die gleichen Werte, beispielsweise für den `id`-Parameter, gewählt werden. So lässt sich der Wert des `id`-Parameters aus dem `Location`-Header, der laut HTTP-Spezifikation beim Erstellen einer Ressource angegeben werden muss, extrahieren und anschließend in anderen Operationen verwenden. Damit lassen sich voneinander abhängige Operationen wie die `POST`-Operation, die eine Ressource erstellt, und die `GET`-Operation, die diese Ressource abruft, beim Testen zuverlässig und mit nur wenigen Testdurchläufen abdecken.

```
1 {
2   "self": "/api/Games/123",
3   "name": "Main Tourney",
4   "playerRefs": [
5     "/api/Players/1",
6     "/api/Players/2"
7   ]
8 }
```

Listing 3.1: Repräsentation einer Spiel-Ressource mit vollständigen Ressourcenidentifikatoren

Da Ressourcenidentifikatoren jedoch nicht nur für den Zugriff auf eine Ressource benötigt werden, sondern ebenso als Hyperlinks verwendet werden, um Verbindungen zwischen Ressourcen zu repräsentieren, bedarf es der Analyse des Anfrage-Bodys, um auch passende Werte beim Testen auszuwählen. Das Listing 3.1 stellt eine Spiel-Ressource dar. Die Repräsentation beinhaltet beginnend ab Zeile vier die Liste der Ressourcenidentifikatoren der Spieler, die an dem Spiel teilnehmen. Die konkrete Benennung der Liste liegt dabei, wie bereits erwähnt, bei den Entwickelnden. Als Benennung dieser Liste könnte statt der Bezeichnung `playerRefs` auch `participants` gewählt werden. Damit sich aus der Benennung Rückschlüsse auf die bestehenden Abhängigkeiten ziehen lassen, müssen also Namenskonventionen beachtet werden. Im Test-Tool können dann mit regulären Ausdrücken die Referenzen auf andere Ressourcen und auf sich selbst erkannt werden. Beispielsweise könnte dann ein Ausdruck, der die Referenzen im Allgemeinen bestimmt, wie:

```
Refs?$
```

und ein Ausdruck, der die Selbstreferenz signalisiert, genutzt werden, wie:

```
^self$
```

```
{
  "id": "123",
  "name": "Main Tourney",
  "playerIds": [
    "1",
    "2"
  ]
}
```

Listing 3.2: Repräsentation einer Spiel-Ressource mit Ids

Der Ansatz zur Analyse von Referenzen, wie sie bei einem RESTful Webservice mit dem Reifegrad *Level 3* vorkommen, lässt sich analog auf einen Webservice mit dem Reifegrad *Level 2* übertragen. Statt mit den vollständigen Referenzen für die Spieler wird dort beispielsweise nur mit `playerIds` wie im Listing 3.2 gearbeitet. Damit beim späteren Testen die Antworten des SUT schnell interpretiert werden können, bietet es sich an, dieselbe Analyse auch für die Antwort-Bodys durchzuführen.

Zusätzlich zu der Zuordnung der einzelnen Parameter zu den Ressourcenklassen bedarf es einer Analyse der Verknüpfungen zwischen den Parametern bzw. Ressourcen. Als Beispiel kann bei dem folgenden Pfad der Parameter `Deposit:@id` nicht alleinstehend betrachtet werden:

```
GET /api/players/{Player:@id}/deposits/{Deposit:@id}
```

Stattdessen muss für eine valide Anfrage für `Deposit:@id` ein Wert gewählt werden, der auf eine `Deposit`-Ressource zeigt, die der durch `Player:@id` referenzierten `Player`-Ressource untergeordnet ist. Zuvor wurde bereits das Erkennen von Eltern-Kind-Beziehungen über die Pfadanalyse dargestellt. Hier kann diese Beziehung und auch ihre Richtung einfach von der Hierarchie des Pfads abgeleitet werden. Bei der Analyse der Anfrage- und Antwort-Bodys ist dies nicht in dieser Art und Weise möglich. So können Ressourcen sowohl ihnen unterge-

```
{
  "self": "/api/Games/123",
  "name": "Main Tourney",
  "participantsRefs": [
    "/api/Players/1",
    "/api/Players/2"
  ],
  "viewerRefs": [
    "/api/Players/3"
  ]
}
```

Listing 3.3: Referenzen mit unterschiedlicher Semantik

ordnete Ressourcen als auch übergeordnete Ressourcen enthalten. Aus diesem Grund wird hier auf das manuelle Eintragen der Richtung der Hierarchie zurückgegriffen. In der Testkonfiguration kann für die der `Player`-Ressource untergeordnete `Deposit`-Ressource die folgende Syntax verwendet werden:

```
Player:@id<Deposit:@id
```

Für eine übergeordnete Ressource müsste stattdessen das Größer-als-Zeichen verwendet werden. Eine Limitation bilden dabei Referenzen auf andere Ressourcen mit unterschiedlicher Semantik, wie es das Listing 3.3 darstellt. Mit dem folgenden Endpunkt können Teilnehmer von der Spieler-Ressource im Listing entfernt werden:

```
DELETE /api/Games/{Game:@id}/participants/{Player:@id}
```

Dafür können aber nur Spielerreferenzen aus der Liste `participantsRefs` genutzt werden, um der Property P8 für existierende Ressourcen zu entsprechen. Im Rahmen dieser Arbeit werden diese Fälle jedoch nicht weiter betrachtet.

Wie sich durch die vorangegangene Erläuterung erkennen lässt, werden im Rahmen der aufgezeigten Limitationen keine drastischen Änderungen der SUT-Schnittstelle benötigt, um die OAS automatisiert

zu analysieren. Essentiell sind die Namenskonventionen für Parameter, die Referenzen ausdrücken. Dabei ermöglicht eine eindeutige, direkte Verknüpfbarkeit dieser Parameter mit dem referenzierten Ressource nicht nur die problemlose Analyse der OAS, sondern hilft genauso auch Menschen, die Schnittstelle unkompliziert zu verstehen. Sollte die Möglichkeit bestehen, auf den Implementierungsprozess des SUT Einfluss zu nehmen, kann die Schnittstelle direkt für das Testen entworfen werden, ohne dass dabei ein zusätzlicher Aufwand entsteht. Trotzdem muss das Ergebnis der automatisierten Analyse noch einmal von den Testenden kontrolliert und angepasst werden, da sich nicht alle Abhängigkeiten zwischen den Ressourcen erkennen lassen.

3.4 Aufbau des Test-Tools

In diesem Abschnitt wird der Aufbau von REST-QA auf Basis der vorangegangen Erläuterungen konstruiert. Grundsätzlich lässt sich das Test-Tool in zwei Teilaufgaben zerlegen. Auf der einen Seite die Analyse der OAS und auf der anderen Seite das Überprüfen der Properties des SUT. Die Abbildung 3.2 stellt den nach außen sichtbaren Datenfluss von REST-QA dar. Im ersten Schritt wird das OAS-Dokument durch das Programm analysiert. Dabei wird eine Testkonfiguration produziert. Dieses Dokument enthält vergleichbar zur OAS die vorhandenen Operationen des SUT, stellt die Verbindungen zwischen

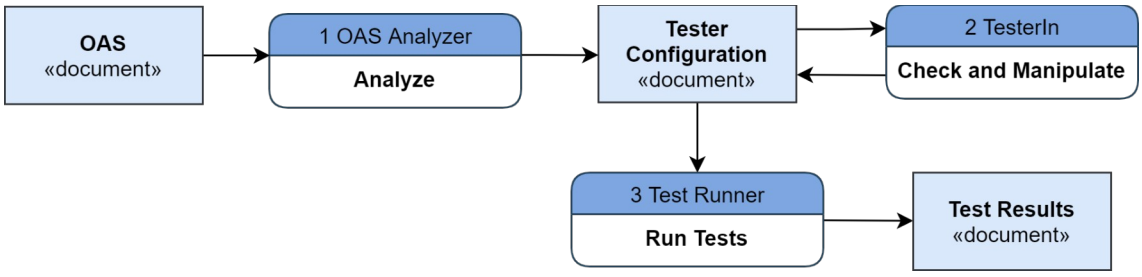


Abbildung 3.2: Datenfluss des Gesamtsystems

den Ressourcen aber explizit dar. Zusätzlich lassen sich weitere testbezogene Konfigurationen setzen, wie beispielsweise die maximale Anzahl an Operationen innerhalb einer Testsequenz. Im zweiten Schritt überprüft der Tester bzw. die Testerin die erzeugte Konfiguration und passt diese an. Wie im Abschnitt 3.3 erläutert wurde, ist dieser Schritt zwingend, da nicht garantiert werden kann, dass die automatisierte Analyse der OAS zu einem korrekten und vollständigen Ergebnis führt. Im letzten Schritt wird das SUT auf Basis der Testkonfiguration von REST-QA getestet. Im Ergebnis wird ein Dokument mit dem Testergebnis ausgegeben. Dieses enthält Informationen über aufgetretene Fehler und die benötigten Schritte zur Reproduktion des Fehlers sowie Daten zur Abdeckung der dokumentierten Statuscodes der einzelnen Operationen. Mit der zuletzt genannten Ausgabe lässt sich eine Aussage über die Qualität des durchgeführten Tests treffen. So entsprechen nicht abgedeckte Statuscodes nicht abgedeckten Programmpfaden im SUT. Die durch die Schritte eins und zwei gewonnene Testkonfiguration kann, solange sich die Schnittstelle des SUT nicht ändert, beliebig oft wiederverwendet werden. Bei der Integration von REST-QA in die Build-Automatisierung muss dann zu einer gegebenen Testkonfiguration nur noch der Test-Schritt ausgeführt und das resultierende Testergebnis interpretiert werden.

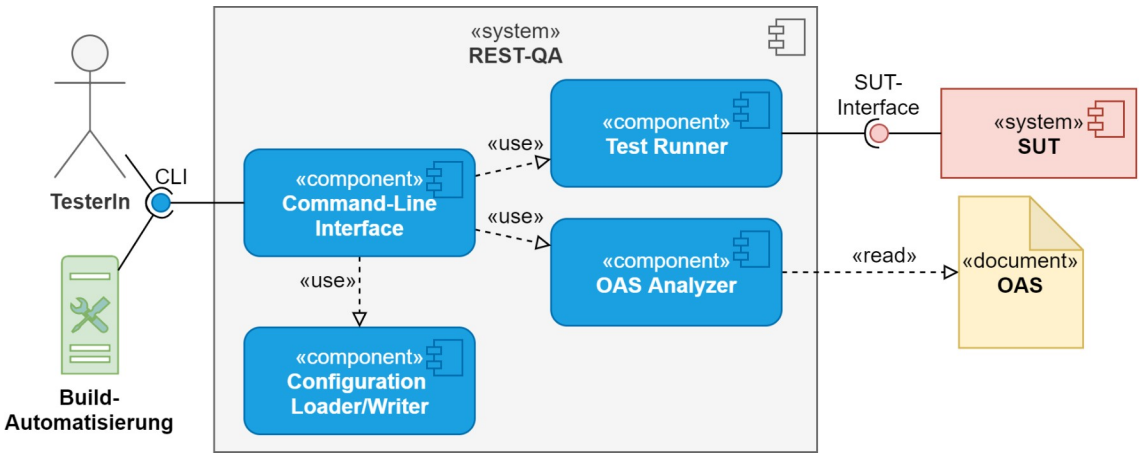


Abbildung 3.3: White-Box-Ansicht REST-QA

Mit Kenntnis des Datenflusses lässt sich nun das Test-Tool in vier zentrale Komponenten wie in Abbildung 3.3 zerteilen. Zunächst wird der *OAS Analyzer* benötigt, um das OAS-Dokument einzulesen und zu analysieren. Die Ausgabe sind die Operationen mit zusätzlichen Kontextinformationen für die Testkonfiguration. Dabei werden die in Abschnitt 3.3 erläuterten Regeln zur Analyse angewandt. Der *Test Runner* ist für die Ausführung der Tests verantwortlich und kommuniziert dafür mit dem SUT. Als Schnittstelle zum Benutzenden und zur Build-Automatisierung dient das *Command-Line Interface*. Über diese Komponente wird der Modus von REST-QA ausgewählt, also das Analysieren der OAS oder Testen des SUT. Der *Configuration Loader/ Writer* wird dafür benötigt, die Testkonfiguration für die Ausgabe zu serialisieren und bei der Eingabe für den Test Runner wieder zu deserialisieren.

3.4.1 Aufbau der Testkonfiguration

Als Nächstes soll der Aufbau des Ergebnisses der Analyse der OAS erörtert werden. Die Testkonfiguration muss dabei den folgenden Anforderungen genügen. Sie muss die ausführbaren Operationen und die Relationen zwischen den Ressourcen und Parametern integrieren, für die Testenden lesbar und anpassbar sein und sich serialisieren lassen. Die größte Herausforderung stellt hierbei die Lesbar- und Anpassbarkeit des Dokuments durch einen Menschen dar. Als Serialisierungssprache wird deshalb YAML genutzt, da das primäre Ziel von YAML die Lesbarkeit durch Menschen ist (vgl. Ben-Kiki et al. 2009). Um die Menge an Informationen zu reduzieren, die die Testenden auf einmal verarbeiten müssen, wird die Testkonfiguration in sieben Gruppen unterteilt, wie es das Listing 3.4 schematisch darstellt. Die erste Gruppe ermöglicht es, den Testdurchlauf zu konfigurieren. Hierzu gehören Einstellungen wie beispielsweise die zu überprüfenden

```
setup: {...}
servers: [...]
resourceClasses:
  Author: &o0
    id: :@id
    name: :name
  Comment: &o2
    id: :@id
    authorId: Author:@id
    message: :message
paths:
  api:
    authors:
      $type: Author[]
      '{Author:@id}':
        $type: Author
    comments:
      $type: Comment[]
      '{Comment:@id}':
        $type: Comment
parameters:
  Get /api/Todo:
    query:
      limit: UNKNOWN_0
securitySchemes: {...}
operations: {...}
```

Listing 3.4: Testkonfiguration (verkürzt)

Properties oder die maximale Anzahl an Testsequenzen, aber auch die Konfiguration der verwendbaren Identitäten, auf die später nochmals eingegangen wird. In der `servers`-Gruppe werden die verwendeten Server spezifiziert. Hier handelt es sich um eine Liste, da es die OAS theoretisch ermöglicht, für jede Operation einen anderen Server zu spezifizieren. Die `resourceClasses`-Gruppe beinhaltet alle gefundenen Klassen, auch solche, die nicht im `Components`-Objekt der OAS global spezifiziert wurden. Dabei werden hier nur die für die Testenden relevante Information wiedergegeben. Dies entspricht der Zuordnung zwischen den Feldern des Objekts und den Ressourcenklassen. Hierbei wird entsprechend dem REST-Architekturstil zwischen Ressourcenrepräsentationen und Ressourcenidentifikatoren unter-

schieden. Die Ressourcenrepräsentationen beginnen direkt mit einem Doppelpunkt, wie der Name des Autors in Zeile sechs, während Ressourcenidentifikatoren mit dem Namen der Ressourcenklasse beginnen, wie `authorId` in Zeile neun. Eine Besonderheit bilden die Referenzen auf die Ressource selbst. Diese Ressourcenidentifikatoren ähneln den Ressourcenrepräsentationen, verwenden aber den geschützten Bezeichner `:@id`, wie es in Zeile fünf dargestellt ist. Die `paths`-Gruppe gibt die Pfade der OAS in Form eines Baums an. Dabei sind die Pfadparameter bereits durch die Zuordnung zu einem Feld einer Ressourcenklasse ersetzt. Der Pfad bis Zeile 15 entspricht also `api/authors/{…}`, wobei der konkrete Name des Parameters `{…}` irrelevant ist, aber stets die Zuordnung zum Feld `Author:@id` besitzt. Die `parameters`-Gruppe spiegelt analog dazu die Zuordnung für solche Parameter wider, die nicht im Pfad vorkommen. Der Teilbaum beginnt dabei jeweils mit der `OperationId` bzw. der Methode und dem Pfad und enthält für jeden Parameter einen Eintrag mit der entsprechenden Zuordnung. Im Listing in Zeile 24 konnte der `limit`-Parameter beispielsweise keiner Ressource zugeordnet werden. Die Gruppe `securitySchemes` entspricht dem `securitySchemes`-Objekt der OAS. Hier werden die vom SUT verwendeten Sicherheitsmecha-

```
1  setup:
2    identityConfiguration:
3    - securityScheme: default_auth
4      scopes:
5      - write:authors
6      - read:authors
7      clientId: CLIENT_ID
8      clientSecret: ********
9  securitySchemes:
10   default_auth:
11     type: OpenIdConnect
12     openIdConnectUrl: http://example.com/.well-known/openid-configuration
```

Listing 3.5: Konfiguration der nutzbaren Identitäten

nismen dokumentiert. Das Listing 3.5 stellt das Zusammenspiel der Konfiguration von Identitäten in der `setup`-Gruppe mit dem spezifizierten Sicherheitsschema in der `securitySchemes`-Gruppe dar. REST-QA unterstützt momentan nur die Verwendung von API-Schlüsseln und OpenId Connect. Im Fall von OpenId Connect muss dabei die Authentifizierung mittels *Client Credentials* erfolgen (vgl. Hardt 2012). Dafür werden `clientId` und `clientSecret` an den Identity Provider übermittelt, um so einen gültigen Token für die Nutzung des SUT zu erhalten. Andere Authentifizierungsmechanismen werden vor-

```yaml
operations:
  /api/Author/{?}:
    Get:
      operationId: GetAuthorById
      parameters:
        path:
        - schema:
            type: integer
            format: int64
      responses:
        200:
          content:
            application/json:
              schema:
                resourceClass: *o0
                type: object
                required:
                - name
                properties:
                  id:
                    type: integer
                    format: int64
                  name:
                    type: string
                additionalPropertiesAllowed: false
    servers:
    - *o1
    security:
    - default_auth:
      - read:authors
```

Listing 3.6: Operation innerhalb der Testkonfiguration (verkürzt)

erst nicht unterstützt, können aber in der Zukunft ergänzt werden. Zusätzlich müssen die Scopes angegeben werden, die diese Identität besitzt, damit die Properties P2 und P3 für die Berechtigungen korrekt überprüft werden können. Es ist anzumerken, dass für einen `securityScheme`-Eintrag mehrere Identitäten mit unterschiedlichen Berechtigungen angegeben werden können. Die letzte Gruppe der Testkonfiguration beinhaltet die Operationen mit ihren Abfrageparametern und den spezifizierten Antworten und ist im Listing 3.6 nochmals explizit dargestellt. Die Syntax ist dabei stark an die der OAS angelehnt. Als Besonderheit besitzen die Pfad-Parameter aber keinen Namen, sondern werden durch ihre Reihenfolge dem entsprechenden Platzhalter im Pfad zugeordnet. Zusätzlich existieren Verknüpfungen zur Ressourcenklasse wie in Zeile 15 und zu den Servern in Zeile 27. Hierbei werden die in YAML integrierten Referenzen, mittels Setzen eines Ankers (vgl. Listing 3.4, Zeile 4) und anschließendem Nutzen des Alias (vgl. Listing 3.6, Zeile 15), verwendet. An den menschlichen Benutzer richten sich vordergründig nur die ersten fünf Gruppen. Um die Fehlerquellen beim manuellen Anpassen zu reduzieren, wurde dort die Konfiguration jeweils nur einfach spezifiziert und wie bereits beschrieben die Menge der Information innerhalb einer Gruppe auf relevante Daten reduziert.

3.4.2 Aufbau der Test-Runner-Komponente

Nachdem der Aufbau der Testkonfiguration beschrieben wurde, soll sich nun der Test-Runner-Komponente gewidmet werden. Diese benötigt die Konfiguration als Eingabe und ist, wie bereits beschrieben, für die Ausführung der eigentlichen Tests verantwortlich. Die Abbildung 3.4 stellt die zentralen Prozesse innerhalb der Komponente dar. Zunächst beginnt die *Test-Maschine*-Komponente mit der Aufbereitung der in der Testkonfiguration spezifizierten Operationen. Auf die

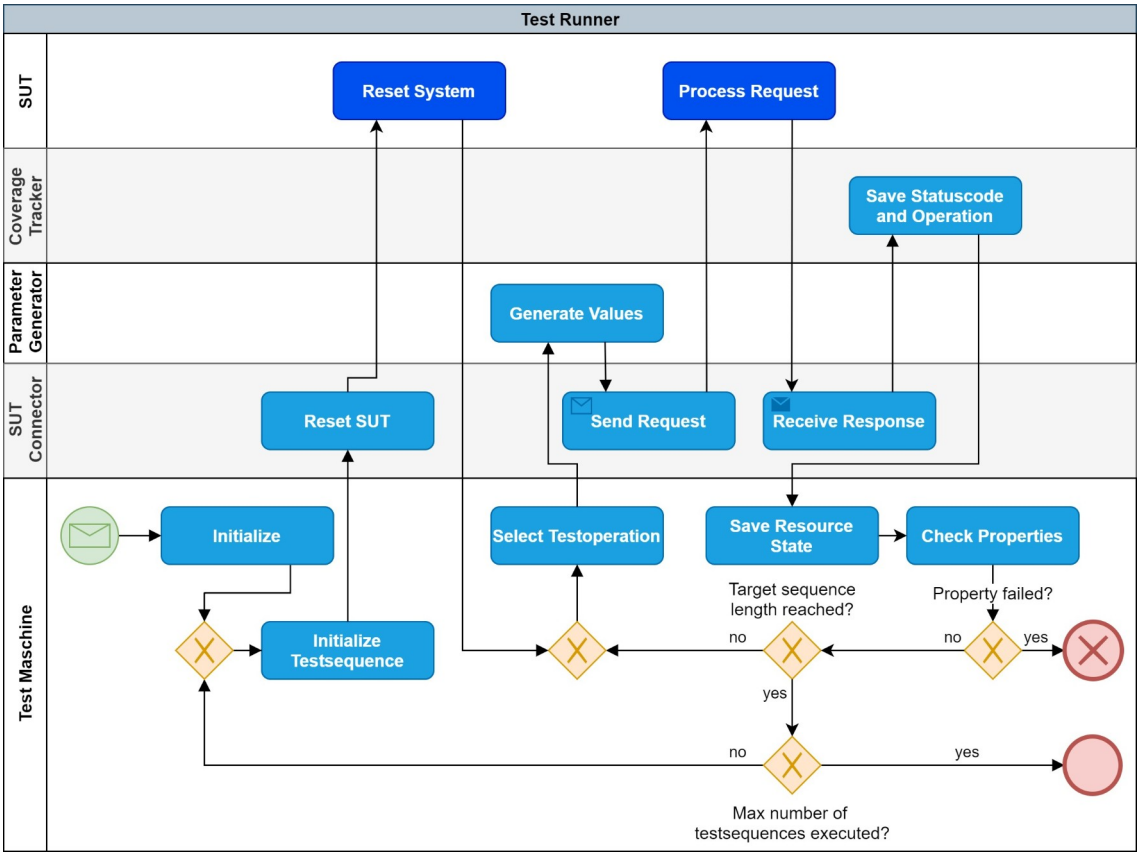


Abbildung 3.4: Prozesse der Test-Runner-Komponente

Details dieser Prozedur wird im Abschnitt 3.5 genau eingegangen. Als Nächstes folgt die Initialisierung einer neuen Testsequenz. Dabei wird das aktuelle Modell vom Zustand des SUT im Test-Tool zurückgesetzt. Im Anschluss wird dann das SUT selbst zurückgesetzt. Bei der Zurücksetzung des SUT wird dabei ein in der Testkonfiguration spezifizierbarer Endpunkt aufgerufen. Die tatsächliche Umsetzung der Zurücksetzung ist vom jeweiligen Testaufbau abhängig und fällt nicht in den Zuständigkeitsbereich von REST-QA. Nachdem sowohl das SUT als auch das Modell des Zustands zurückgesetzt wurden, wird die erste Testoperation ausgewählt. Auch hier werden die Details im Abschnitt 3.5 erläutert. Für die gewählte Testoperation werden dann anhand der JSON-Schemata gültige Werte für die Parameter der Operation generiert. Mit der Spezifikation der Operation und den generierten Werten wird nun von der *SUT-Connector*-Komponente

eine syntaktisch valide HTTP-Anfrage generiert und an das SUT gesendet. Das SUT bearbeitet daraufhin die Anfrage und sendet die Antwort zurück an REST-QA. Hier wird diese dann an die *Coverage-Tracker*-Komponente weitergegeben. Diese Komponente ermöglicht es am Ende des Tests, die Abdeckung der Statuscodes pro Operation durch die Testfälle zu ermitteln. Danach wird das Modell mit dem aktuellen Zustand der Ressource, auf die sich die HTTP-Anfrage bezieht, aktualisiert. Als Beispiel kann eine Ressource als bekannt hinterlegt werden, nachdem sie durch eine erfolgreiche POST-Anfrage erstellt wurde. Im Anschluss werden die Properties geprüft. Sollte einer der Properties falsch sein, wird der gesamte Test an dieser Stelle beendet und die Testsequenz zurückgegeben, die den Fehler erzeugt hat. Andernfalls wird überprüft, ob die angestrebte Anzahl an Testoperationen in der aktuellen Testsequenz noch nicht erreicht ist. Ist das der Fall, wird die nächste Testoperation ausgewählt. Sollte stattdessen die Testsequenz bereits abgeschlossen sein, wird nun überprüft, ob weitere Testsequenzen zu generieren sind. Falls ja, wird eine neue Testsequenz initialisiert, falls nein, wurde der Test erfolgreich abgeschlossen.

Die Abbildung 3.5 stellt nochmals die Test-Runner-Komponente mit den internen und externen Abhängigkeiten dar. Die Test-Maschine-Komponente ist dabei vorrangig für die Steuerung des Testprozesses

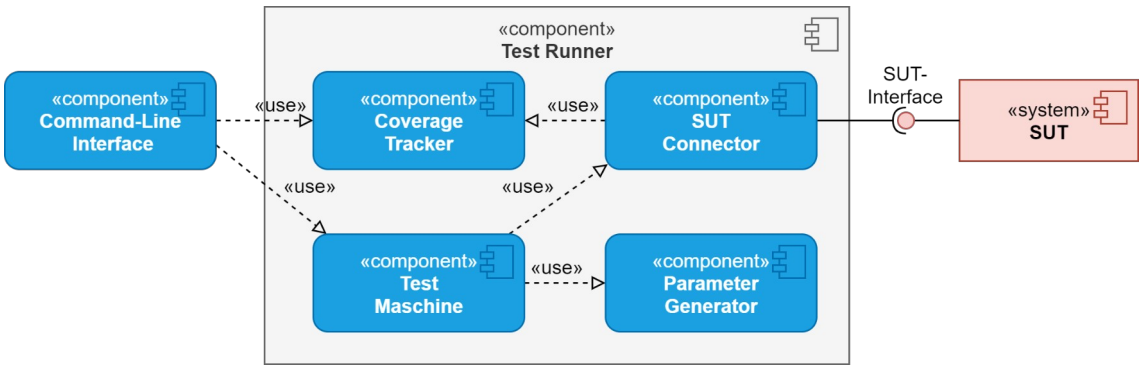


Abbildung 3.5: White-Box-Ansicht Test Runner

verantwortlich. Die Command-Line-Interface-Komponente ist für den Abruf der Analyse der Statuscodeabdeckung der einzelnen Operationen von der Coverage-Tracker-Komponente und das Generieren des Dokuments mit den Testergebnissen verantwortlich.

3.5 Generieren von Testsequenzen und -werten

In diesem Abschnitt wird die Generierung von Testsequenzen und die der Wertebelegung für die Parameter einer Operation detaillierter erläutert. Dabei wird zunächst die Initialisierung der Test-Maschine-Komponente betrachtet. Aus der eingegebenen Testkonfiguration werden in diesem ersten Schritt die verfügbaren Operationen extrahiert. Dabei werden aus den Parametern und auch aus dem Anfrage-Body mittels der in der Konfiguration spezifizierten Zuordnungen zwischen den Parametern bzw. JSON-Schemata und Ressourcen die einzigartigen Eingabeparameter ermittelt. Als Veranschaulichung soll das Listing 3.7 dienen. Hier wird `Player:@id` sowohl im Pfad der Anfrage als auch im Body genutzt. Um eine valide Anfrage an das SUT zu stellen, müssen beide Parameter denselben Wert besitzen. Für die einzigartigen Eingabeparameter werden darüber hinaus zusätzliche Eigenschaften abgeleitet. So werden die Beziehungen aus der in der Testkonfiguration definierten Hierarchie hinterlegt, da diese Information bei der späteren Parametergenerierung berücksichtigt werden

```
1 PUT /api/players/123 HTTP/1.1
2 Host: example.com:80
3 Content-Length: 32
4 Content-Type: application/json
5
6 {
7   "id": 123,
8   "budget": 500
9 }
```

Listing 3.7: PUT-HTTP-Anfrage (gekürzt)

muss. Des Weiteren wird der Parametertyp gespeichert, hier wird zwischen Ressourcenrepräsentation, einer Referenz auf eine andere Ressource und einer Selbstreferenz unterschieden. Bei einer Selbstreferenz wird zusätzlich überprüft, ob der Parameter für die Erstellung einer neuen Ressource benötigt wird, wie es bei POST- oder PUT-Anfragen der Fall sein kann. Diese Besonderheit kann sich dann beim späteren Generieren der Werte zu Nutze gemacht werden. Zuletzt wird zu jedem einzigartigen Eingabeparameter gespeichert, ob dieser Bestandteil des Ressourcenidentifikators ist. Hier werden, abhängig von der Konfiguration, die Query-Parameter ignoriert oder mit eingeschlossen. Mit dieser Information werden nun die Operationen in die sog. Testoperationen überführt. Für jede der in Abschnitt 3.2 beschriebenen Zustände einer Ressource existiert eine Testoperationsklasse mit den entsprechenden Properties. Zur Wiederholung, es wird zwischen existierenden, gelöschten und unbekannten Ressourcen unterschieden. Jede in der Testkonfiguration spezifizierte Operation wird nun in eine oder mehrere Testoperationen konvertiert. Dies soll anhand der folgenden Operation verdeutlicht werden:

```
GET /api/players/{Player:@id}
```

Zunächst lässt sich diese Operation für eine existierende `Player`-Ressource ausführen, also wird eine Testoperation für existierende Ressourcen hinterlegt. Da die Operation `Player:@id` für die Ressourcenidentifikation benötigt, kann hierfür auch gezielt ein unbekannter Wert oder der Wert einer gelöschten `Player`-Ressource verwendet werden. Dementsprechend werden zusätzlich eine Testoperation für unbekannte und eine Testoperation für gelöschte Ressourcen hinzugefügt. Im Gegensatz dazu kann die folgende Operation nur in eine Testoperation für existierende Ressourcen konvertiert werden:

```
GET /api/players
```

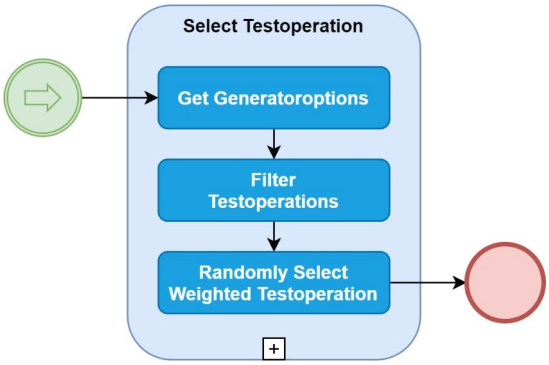


Abbildung 3.6: Prozesse zur Auswahl einer Testoperation

Dies hat den Hintergrund, dass es hier keinen Parameter gibt, der den Wert für eine Referenz auf eine unbekannte oder gelöschte Ressource annehmen kann. Folglich kann für jede Operation mindestens die Testoperation für existierende Ressourcen abgeleitet werden.

Nach dieser initialisierenden Prozedur folgen die Initialisierung der Testsequenz und anschließend die Prozesse entsprechend der Abbildung 3.6. Bei Abbildung handelt es sich um die Aufschlüsselung des Prozesses zur Auswahl einer Testoperation innerhalb der Test-Maschine-Komponente in die drei folgenden Teilprozesse. Zunächst werden zu jeder Testoperation alle möglichen *Generatoroptionen* berechnet. Eine Generatoroption bestimmt dabei für jeden einzigartigen Eingabeparameter einen der drei folgenden Modi für den Generator: es kann entweder ein rein zufälliger Wert im Rahmen des JSON-Schemas gewählt werden, es kann ein zwingend unbekannter Wert verlangt werden oder es kann ein konstanter Wert verwendet werden. Dies hat den Hintergrund, dass die Testoperationen vom Zustand der Ressourcen abhängig sind und somit nur bestimmte Wertebelegungen zu den beabsichtigten Antworten führen können. Im nächsten Schritt werden alle Testoperationen herausgefiltert, für die im aktuellen Zustand des SUT keine validen Generatoroptionen existieren. Ergo können diese Testoperationen momentan nicht ausgeführt werden. Da sich der Zustand des SUT im Lauf einer Testsequenz

ändert, verändert sich auch die Menge der verfügbaren Testoperationen. So steht beispielsweise die `DELETE`-Operation für eine Ressource erst dann zur Auswahl, nachdem diese durch eine `POST`-Operation angelegt wurde. Die Selektion der nächsten Testoperation erfolgt dabei zufällig mit einer unterschiedlichen Gewichtung der einzelnen Operationen. Dass die zufällige Auswahl der Testoperationen geeignet ist, um RESTful Webservices zu testen, hat bereits Atlidakis et al. 2019 ermittelt. Beim Vergleich der Algorithmen *Breadth-first search* (BFS), BFS-Fast und *RandomWalk*, konnte der letztgenannte in unterschiedlichen Tests zwar nicht immer die höchste Quelltext-Abdeckung erreichen, war den anderen beim Auffinden von Fehlern jedoch stets überlegen. Bei der Gewichtung der Testoperationen werden zwei Kriterien angewandt. Zunächst werden die Testoperationen entsprechend ihrer HTTP-Methode unterschiedlich gewichtet. Viglianisi et al. 2020 schlagen dabei eine Priorisierung in der Reihenfolge: POST, GET, PUT, DELETE vor, die in dieser Form von REST-QA adaptiert wird. Darüber hinaus werden Testoperationen nach dem vorausgesetzten Zustand der Ressource gewichtet. Die höchste Priorität haben hierbei Operationen auf existierenden Ressourcen, gefolgt von Operationen auf gelöschten Ressourcen. Die niedrigste Gewichtung haben Operationen auf unbekannten Ressourcen. Dies hat den Hintergrund, dass die Wahrscheinlichkeit höher ist, dass Operationen auf existierenden Ressourcen den Zustand des SUT verändern, während Operationen auf nicht existierenden oder gelöschten Ressourcen meist zu einem trivialen Statuscode `404` (Not Found) bzw. Statuscode `410` (Gone) führen. Dennoch sind die zweitgenannten Operationen in der Lage, zusätzliche Fehler zu finden. Während der Entwicklung von REST-QA konnte beispielsweise eine Inkonsistenz der Datenbank nach dem Löschen einer Ressource entdeckt werden. Dabei wurde zuerst (1) eine Blog-Ressource angelegt und dazu (2) eine untergeordnete

Kommentar-Ressource hinzugefügt, woraufhin (3) die Blog-Ressource aus (1) wieder gelöscht wurde. Im Anschluss wurde (4) eine neue Blog-Ressource mit demselben Identifikator wie aus (1) erstellt. (5) Bei der Abfrage der folgenden Operation:

```
GET /api/blogs/{Blog:@id}/comments/{Comment:@id}
```

mit den Identifikatoren aus (1) und (2) wurde der Statuscode `200` (OK) zurückgegeben. Dabei hätte die Kommentar-Ressource aus (2), die sich auf die ursprünglichen Blog-Ressource aus (1) bezieht, mit dem Löschen des Blogs in (3) ebenfalls gelöscht werden sollen. Dieser Fehler wurde auf Grund der Property P7 für gelöschte Ressourcen entdeckt. Im Abschnitt 4.3 wird der Einfluss der zustandsabhängigen Properties auf den Test nochmals diskutiert.

```
1  IEnumerable<GeneratorOption> GetGeneratorOptions(TestModel model)
2  {
3      var baseOpt = new GeneratorOption();
4      var needs = Operation.GetNeeds();
5
6      foreach (var p in Operation.UniqueParameters.Except(needs))
7          baseOpt.Add(p.Mapping,
8              p.IsSelfRef ? GenMode.RequireUnknown : GenMode.Random);
9
10     if (!needs.Any())
11         return new List<GeneratorOption> { baseOpt };
12
13     return model.GetMatchingParameterSubset(needs)
14         .Where(set => set.All(id => !id.IsDeleted));
15         .Select(parameters =>
16         {
17             var option = new GeneratorOption(baseOpt);
18             foreach (var p in parameters)
19                 option.AddConstant(p.Mapping, p.Valuevalue);
20
21             return option;
22         });
23 }
```

Listing 3.8: Ermittlung der Generatoroptionen (C#)

Daran anknüpfend soll nun die Prozedur zur Ermittlung der Generatoroptionen für Testoperationen auf existierende Ressource, wie im Listing 3.8, im Detail betrachtet werden. Zuerst werden in Zeile vier die benötigten einzigartigen Eingabeparameter für die Operation erörtert. Das sind all die Parameter, bei denen es sich entweder um eine Referenz oder um eine nicht frei wählbare Selbstreferenz handelt. In den Zeilen sechs bis acht wird für alle einzigartigen Eingabeparameter, die nicht zu den benötigten zählen, der Zufallsgeneratormodus gewählt. Eine Ausnahme bilden Selbstreferenzen. Hier wird verlangt, dass der Wert zwingend unbekannt ist. Dies ist relevant, da beim Erstellen einer Ressource der selbstreferenzierende Ressourcenidentifikator einzigartig sein muss. Sollten keine benötigten Eingabeparameter existieren, wird in Zeile elf eine Generatoroption zurückgegeben, bei der alle Parameter zufällig mit bzw. ohne die Verwendung von bekannten Werten generiert werden. Dies wäre beispielsweise bei der folgenden Operation der Fall:

```
GET /api/players/
```

Andernfalls werden in der 13. Zeile für den aktuellen Zustand des SUT alle Kombinationen an Wertebelegungen zurückgeben, die den Constraints der benötigten Eingabeparameter genügen. Hierbei werden auch die bereits beschriebenen Abhängigkeiten unter den Parametern beachtet. In der darauffolgenden Zeile werden dann alle Kombinationen herausgefiltert, bei denen mindestens ein Wert als gelöscht markiert ist. In den Zeilen 15 bis 22 wird nun für jede der verbleibenden Kombinationen eine Generatoroption erstellt. Dabei wird für die Eingabeparameter jeweils der existierende Wert als Konstante hinterlegt.

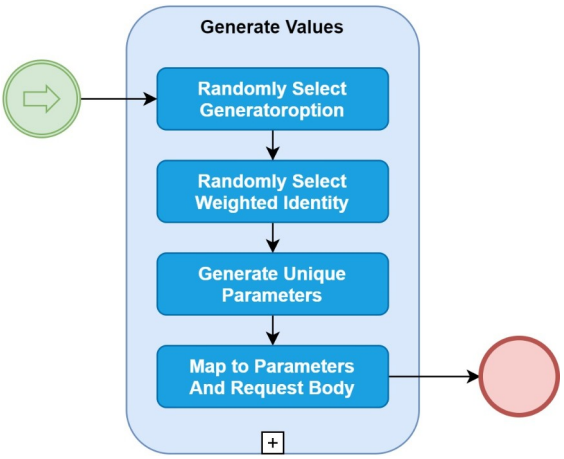


Abbildung 3.7: Prozesse zur Generierung von Testwerten

Nachdem eine Testoperation ausgewählt wurde, folgt die Generierung der Werte für die Parameter der Operation. Die Abbildung 3.7 stellt dazu die vier Teilprozesse innerhalb der Parameter-Generator-Komponente dar. Aus der Menge von berechneten Generatoroptionen für die Testoperation wird eine zufällig ausgewählt. Im nächsten Schritt wird zufällig ausgewählt, ob die Operation mit einer validen oder invaliden Identität ausgeführt wird. Eine valide Identität hat in diesem Kontext alle Berechtigungen, die zum Ausführen der Operation vom SUT benötigt werden. Die Wahrscheinlichkeit, dass eine valide Identität gewählt wird, lässt sich in der Testkonfiguration einstellen. Da Operationen, die mit invaliden Identitäten ausgeführt werden, im Normalfall zum Statuscode `403` (Forbidden) führen, ändert sich der Zustand des SUT durch solche Operationen nicht. Deshalb ist es sinnvoll, REST-QA so zu konfigurieren, dass primär Anfragen mit validen Identitäten ausgeführt werden. Im dritten Schritt werden für jeden einzigartigen Eingabeparameter Werte entsprechend der Generatoroption gewählt. Im letzten Schritt werden die generierten Werte den Parametern und dem Anfrage-Body der Operation zugeordnet. Die ausgewählte Testoperation kann nun in Kombination mit den generierten Werten für die Parameter in eine HTTP-Anfrage konvertiert und von der SUT-Connector-Komponente an das SUT gesendet werden.

4 VALIDIERUNG

In diesem Abschnitt wird die Validierung des konzipierten Test-Tools erläutert. Dazu wird mittels REST-QA die GitLab-Schnittstelle überprüft, vergleichbar mit den Arbeiten von Atlidakis et al. 2019 und Karlsson et al. 2020. GitLab beschreibt sich selbst als offene *DevOps*-Plattform und ermöglicht u. a. die Erstellung, das Planen, die Testintegration und die Veröffentlichung von Software-Projekten (vgl. GitLab o. J. a). Neben der Vergleichbarkeit zu den genannten Arbeiten eignet sich GitLab für die Validierung von REST-QA, da es zum einen eine komplexe Schnittstelle mit diversen Abhängigkeiten unter den Ressourcen besitzt und zum anderen die *Community Edition* als Open-Source-Projekt frei zur Verfügung steht. Als Erstes werden dabei der Testaufbau und die Durchführung des Tests beschrieben. Im Anschluss werden dann die Ergebnisse dargestellt. Im letzten Abschnitt werden diese diskutiert und Bezug auf die Konzeption aus Abschnitt 3 genommen.

4.1 Testaufbau und Durchführung

Für die Durchführung des Tests wird die *Community Edition* von GitLab mit der Image-Version *13.11.2ce.0* als alleinstehender Docker-Container mit der Standardkonfiguration betrieben (vgl. GitLab o. J. b; Docker 2021). Als Ausführungsumgebung dient ein Windows-PC mit 16 GB Arbeitsspeicher, einem i7-5820K-CPU mit 3.30 GHz, sechs physischen Kernen und zwölf logischen Prozessoren. Als Speicher für die persistierten Daten wird ein *Solid State Drive* (SSD) mit einer Transferrate von 540 MB/s beim Lesen und 520 MB/s beim Schreiben verwendet. Bei der Ausführung der Tests werden REST-QA und das SUT auf demselben System betrieben. Um sicherzustellen, dass sich das SUT zu Beginn einer Testsequenz stets im selben Zustand

befindet, wird vor der Durchführung jeder Sequenz ein neuer Container gestartet. Der Container wird dann am Ende einer Testsequenz mitsamt seiner persistierten Daten gelöscht.

Bevor mit dem eigentlichen Testen begonnen werden konnte, musste zunächst eine *OpenAPI Specification* (OAS) entwickelt werden, da GitLab selbst keine vollständige OAS besitzt. Die Entwicklung der OAS erfolgte dabei iterativ. Zunächst wurde eine Spezifikation auf Grundlage der Online-Dokumentation abgeleitet (vgl. GitLab o. J. c). Dabei wurde nur ein Ausschnitt der gesamten Schnittstelle betrachtet, angelehnt an die in der Dokumentation enthaltenen `Projects`-Ressource und `Branches`-Ressource. Die entsprechenden Operationen sind in Tabelle 4.1 dargestellt. Im nächsten Schritt wurde aus der OAS mit Hilfe von REST-QA die Testkonfiguration generiert. An dieser mussten dann mehrere Anpassungen vorgenommen werden. Zunächst wurden die Eltern-Kind-Beziehungen im Datenmodell gepflegt. Danach wurden die Zuordnungen, die REST-QA nicht auto-

Methode	Pfad
GET	/projects
POST	/projects
GET	/projects/{id}
PUT	/projects/{id}
DELETE	/projects/{id}
POST	/projects/{id}/star
POST	/projects/{id}/unstar
GET	/projects/{id}/starrers
POST	/projects/{id}/archive
POST	/projects/{id}/unarchive
POST	/projects/{id}/restore
GET	/projects/{id}/repository/branches
POST	/projects/{id}/repository/branches
GET	/projects/{id}/repository/branches/{branch}
DELETE	/projects/{id}/repository/branches/{branch}
POST	/projects/user/{user_id}
GET	/user
GET	/users/{user_id}/projects
GET	/users/{user_id}/starred_projects

Tabelle 4.1: Liste der ausgewählten Operationen

matisch erkannt hat, korrigiert. Dies betraf beispielsweise die `creator_id` der `Project`-Ressource, die nicht der `User`-Ressource zugeordnet wurde, oder auch die Zuordnung des Parameters `ref`, der eine Referenz auf einen existierenden *Branch* oder *Commit* darstellt. Zusätzlich wurden Parameter, die das Filtern einer Ressource über Freitext ermöglichten, wie der Parameter `search` oder `with_programming_language`, entfernt. Dies hat den Hintergrund, dass bei einer zufälligen Belegung dieser Parameter die Wahrscheinlichkeit, eine nicht triviale Antwort vom SUT zu erhalten, sehr gering ist. Aus demselben Grund werden die Parameter für die Nutzung einer Vorlage bzw. eines Imports bei der Erstellung von Projekten gestrichen. Dies hat zur Folge, dass REST-QA keine Kenntnis von diesen Parametern hat und somit nur Anfragen ohne diese Parameter generiert. Der Generator für die Testoperationen wurde so konfiguriert, dass optionale Parameter mit einer Wahrscheinlichkeit von 0,5 ignoriert werden. Für Parameter, die den Wert `null` annehmen können, wurde die Wahrscheinlichkeit ebenfalls auf 0,5 gesetzt, dass diese mit `null` statt eines entsprechenden Werts belegt werden. Da das Berechtigungskonzept von GitLab nicht Claim-basiert im Sinne der Definition dieser Arbeit ist, wird die Wahrscheinlichkeit, eine Anfrage mit invalider Identität zu generieren, auf 0 gesetzt. Als ein Beispiel, welches gegen das in der Arbeit beschriebene Berechtigungskonzept verstößt, lässt sich die Möglichkeit, Projekte als *öffentlich* zu deklarieren, nennen. In diesem Fall ist es möglich, auf diese Projekte ohne Authentifizierung, also mit einer invaliden Identität, zuzugreifen. Darüber hinaus werden für die Generierung von Freitext-Parametern nur alphanumerische Zeichen genutzt.

Mit dieser Testkonfiguration wurden nun mehrere Tests ausgeführt. Dabei wurde die zugrundeliegende OAS immer wieder angepasst. Der primäre Grund dafür ist, dass die offizielle Online-Dokumentation von GitLab das Verhalten nicht vollständig dokumentiert. Dementsprechend hat REST-QA stets einen Verstoß gegen die Property P1, dass die Antwort des SUT nicht in der OAS enthalten ist oder nicht mit dieser übereinstimmt, festgestellt. Als Beispiel lässt sich das Erstellen einer Branch-Ressource für ein archiviertes Projekt anführen. Bei archivierten Projekten ist die Bearbeitung der Branch-Ressource nicht möglich und das SUT antwortet stets mit dem Statuscode `403` (Forbidden). Dieser Umstand wird weder im Kontext des Archivieren eines Projekts noch beim Erstellen einer Branch-Ressource dokumentiert. Selbiges gilt auch für die Spezifikation der Anfrage-Parameter. Bei Freitext-Parametern fehlt beispielsweise die minimal oder maximal erlaubte Zeichenlänge. Bei `keep_n` innerhalb des Parameters `container_expiration_policy_attributes` sind nur bestimmte Zahlen bei der Eingabe erlaubt, ohne dass dies in der Dokumentation erwähnt wird. Deshalb mussten, unabhängig von REST-QA, empirisch alle möglichen Belegungen (1, 5, 10, 25, 50, 100) ermittelt werden. Ein weiterer Grund, die OAS bzw. Testkonfiguration zu bearbeiten, war das Vermeiden von bereits gefundenen Fehlern, die im Abschnitt 4.2 erläutert werden. Mit den so gewonnenen Erkenntnissen wurde die OAS nach und nach weiter angepasst, wobei sich die finale Version im Anhang 1 einsehen lässt.

Während der Verfeinerung der OAS wurde die Länge der Tests immer weiter gesteigert. Da das Starten eines Containers bis zu dem Zeitpunkt, an dem das SUT Anfragen verarbeiten kann, ca. drei Minuten in Anspruch nimmt, wurden möglichst lange Testsequenzen mit einer geringen Gesamtanzahl an Testsequenzen ausgeführt. Als maximale Sequenzlänge wurde 1000 festgelegt mit bis zu 100 Testsequenzen.

In Summe wurden also 100.000 Anfragen an das SUT während eines vollständigen Tests gesendet. Da REST-QA und das SUT auf demselben System betrieben werden, wurde REST-QA so angepasst, dass es Antworten vom SUT mit dem Statuscode `502` (Bad Gateway), die auf die temporäre Nichtverfügbarkeit eines nachgelagerten Endpunkts hinweisen, nicht als Fehler interpretiert. Stattdessen wird dann die Anfrage wiederholt.

4.2 Ergebnisse

Bei der Betrachtung der Ergebnisse der durchgeführten Tests soll zunächst die durch REST-QA erreichte Abdeckung der Schnittstelle diskutiert werden. Im Folgenden wird die Ausgabe eines Tests mit einer Sequenzlänge von 1000 und einer Sequenzanzahl von 100 betrachtet. Die Tabelle 4.2 ordnet dabei jeder Operation alle Status-

Methode	Pfad	Statuscodes
GET	/projects	200
POST	/projects	201, 400
GET	/projects/{id}	200, 404
PUT	/projects/{id}	200, 400, 404
DELETE	/projects/{id}	202, 404
POST	/projects/{id}/star	201, 304, 404
POST	/projects/{id}/unstar	201, 304, 404
GET	/projects/{id}/starrers	200, 404
POST	/projects/{id}/archive	201, 404
POST	/projects/{id}/unarchive	201, 404
POST	/projects/{id}/restore	404
GET	/projects/{id}/repository/branches	200, 404
POST	/projects/{id}/repository/branches	201, 403, 404
GET	/projects/{id}/repository/branches/{branch}	200, 404
DELETE	/projects/{id}/repository/branches/{branch}	204, 400, 403, 404
POST	/projects/user/{user_id}	201, 400, 404
GET	/user	200
GET	/users/{user_id}/projects	200, 404
GET	/users/{user_id}/starred_projects	200, 404

Tabelle 4.2: Erzeugte Statuscodes für jede Operationen

codes zu, die vom SUT während des Tests zurückgegeben wurden, exklusive der `5XX`-Statuscodes. Zunächst lässt sich hier erkennen, dass für alle Operationen bis auf eine Ausnahme ein Statuscode zurückgegeben wurde, der eine erfolgreiche Ausführung indiziert. Um dieses Ergebnis in den Kontext zum benötigten Zustand der Ressourcen zu setzen, sollen exemplarisch die einzelnen Schritte zur erfolgreichen Löschung einer Branch-Ressource veranschaulicht werden. Dazu muss REST-QA zunächst (1) eine Projekt-Ressource erstellen und dabei das *Repository* über den `init_with_readme`-Parameter initialisieren, da ansonsten keine Repository- und demnach auch keine Branch-Ressource angelegt wird. Im nächsten Schritt muss REST-QA (2) den Namen der erstellten Branch-Ressource mittels der Operation zum Auslesen der Branches abfragen. Da der Standard-Branch selbst nicht gelöscht werden kann, muss zunächst (3) eine weitere Branch-Ressource erstellt werden. Dafür wird eine Referenz auf eine existierende Branch- oder *Commit*-Ressource benötigt. Die in (3) erstellte Branch-Ressource kann nun (4) von REST-QA gelöscht werden und somit den Statuscode `204` erzeugen. Die einzige Operation, für die keine erfolgreiche Antwort generiert werden konnte, bildet die Operation zum Wiederherstellen eines Projekts. Die Ursache liegt darin begründet, dass GitLab seit der Version 13.2 eine sofortige statt verzögerte Löschung des Projekts durchführt (vgl. GitLab o. J. c). Da in der OAS kein Endpunkt definiert ist, der dieses Verhalten beeinflussen kann, ist es für REST-QA nicht möglich, ein Projekt erfolgreich wiederherzustellen. Als Zweites lässt sich in der Tabelle erkennen, dass für alle Operationen mit mindestens einem Pfadparameter der Statuscode `404` (Not Found) erzeugt werden konnte. Da REST-QA stets versucht, valide Anfragen zu generieren, ist dies auf die zustandsabhängigen Testoperationen für unbekannte und gelöschte Ressourcen zurückzuführen. Bei diesen Testoperationen

werden gezielt unbekannte Referenzen bzw. Referenzen auf gelöschte Ressourcen verwendet und als Antwort der Statuscode `404` (Not Found) oder auch `410` (Gone) erwartet. Der zweitgenannte Statuscode findet aber im Fall von GitLab keine Verwendung. Obwohl REST-QA, wie bereits erläutert, darauf ausgelegt ist, ausschließlich valide Anfragen zu generieren, wurden trotzdem im Kontext der Erstellung und Bearbeitung einer Projekt-Ressource sowie bei der Löschung einer Branch-Ressource der Statuscode `400` (Bad Request) vom SUT zurückgegeben. Im ersten Fall lässt sich dies auf einen Fehler in der Validierung oder aber eine inkorrekte Fehlermeldung zurückführen. Für den Pfad wird beispielsweise der folgende 235 Zeichen lange Wert angeben:

```
zRROnLTpaStD1JKNKE6ehIDRhmx8MsXy56o68rt7GhiPTRTMeqmkq3htg20bVl0UG
FngZ7NWFtCpOzqOe0QYF40qFarNF8FQD7bV9i8PTJkJGLSjvyhknAoM2vOvV7LCvf
ycuppzZi55r0Ic5NrffPbhjM6Ka1ai2kMWPlgC6ZbL47p0cbpMTBO7r63s33VqAts
UJ1cwbsCRFGQPQXkzi0Bxmy7uEa7uwkNXZyfSEIc
```

Jedoch antwortet das SUT sowohl beim Erstellen als auch beim Bearbeiten mit der Fehlernachricht:

```
HTTP/1.1 400 Bad Request
Content-Length: 93
Content-Type: application/json

{"message":{"path":["is too long (maximum is 255
characters)"],"name":[],"limit_reached":[]}}
```

Das heißt, obwohl der Wert des Pfad-Parameters den in der Fehlernachricht beschriebenen Anforderungen genügt, wird die Anfrage abgelehnt. Der Grund für die Ablehnung liegt darin, dass die Kombination aus dem Pfad der Namespace-Ressource und dem Pfad der Projekt-Ressource das Maximum von 255 Zeichen nicht übersteigen darf. Da der Pfad der Namespace-Ressource dynamisch ist, lässt sich dieser Zusammenhang jedoch nicht über das JSON-Schema in der OAS definieren. Beim Löschen einer Branch-Ressource indiziert der

Statuscode `400` (Bad Request) hingegen, dass versucht wurde, den *Default Branch* zu löschen. Hierbei handelt es sich also weniger um eine falsche Eingabe, sondern vielmehr um eine verbotene Operation. Selbiges gilt für den Statuscode `403` (Forbidden) beim Erstellen und Löschen einer Branch-Ressource. Dieser Statuscode wird zurückgegeben, wenn versucht wird, die Branches eines archivierten Projekts zu verändern. Für die beiden Operationen, bei denen der Statuscode `304` (Not Modified) zurückgegeben wird, zeigt eben dieser Statuscode an, dass sich die Ressource durch die Bearbeitung nicht verändert hat. Anhand der dargestellten Tabelle und den beschriebenen Erläuterungen lässt sich die Aussage treffen, dass das entwickelte Test-Tool in der Lage ist, die Funktionalitäten von komplexen Schnittstellen mit voneinander abhängigen Ressourcen ausführlich zu testen. REST-QA deckt dabei jedoch nur die Validierung der Eingaben mit Werten, die den Bedingungen der Spezifikation genügen, ab. Es wird also nicht überprüft, wie das SUT mit gezielt falsch aufgebauten Anfragen umgeht.

Nach der Betrachtung der durch REST-QA erzielten Abdeckung folgt nun die Beschreibung der gefundenen Fehler. Bei der Durchführung der Tests konnte ein Fehler, den bereits Karlsson et al. 2020 gefunden haben, reproduziert werden (vgl. Karlsson 2019 a). (F1): Bei diesem zustandslosen Fehler wird von GitLab der Statuscode `500` (Internal Server Error) zurückgegeben, wenn für den `per_page`-Parameter beim Auslesen von Projekten der Wert `0` angeben wird, wie:

```
GET /users/1/starred_projects?page=125&per_page=0
```

Um diesen Fehler in zukünftigen Tests auszuschließen, wurde in der OAS das Minimum von `1` für den `per_page`-Parameter definiert. Da sowohl für das Erstellen als auch für das Aktualisieren von Projekt-Ressourcen zu Beginn der Tests dasselbe Schema verwendet wurde,

wurden die folgenden beiden Fehler entdeckt. (F2): Wenn beim Erstellen eines Projekts der Parameter `ci_default_git_depth` angegeben wird, dann wird die Operation abgebrochen und das SUT antwortet mit dem Statuscode `500` (Internal Server Error). Dieser Fehler wurde bereits zuvor manuell bei der Bearbeitung eines anderen Fehlers von Troschinetz 2020 entdeckt. (F3): Beim Erstellen tritt dasselbe Verhalten auf, wenn der Parameter `default_branch` in der Anfrage enthalten ist. Dieser Fehler tritt aber nur auf, wenn das Projekt für einen bestimmten Nutzer erstellt wird:

```
POST /projects/user/1 HTTP/1.1
Content-Type: application/json
Content-Length: 49

{"name": "random-name","default_branch": "main"}
```

Bei der einfachen Erstellung eines Projekts hingegen kann `default_branch` erfolgreich verarbeitet werden:

```
POST /projects HTTP/1.1
Content-Type: application/json
Content-Length: 49

{"name": "random-name","default_branch": "main"}
```

Die JSON-Schemata der OAS wurden daraufhin entsprechend angepasst. Des Weiteren konnten auch zwei zustandsabhängige Fehler gefunden werden. (F4): Wenn bei der Erstellung eines neuen Projekts ein *Tag* mit mehr als 255 Zeichen angegeben wird, so kann das Projekt erfolgreich erstellt werden. Bei der anschließenden Abfrage der Branches dieses Projekts wird dann aber der Statuscode `500` (Internal Server Error) zurückgegeben. (F5): Der zweite Fehler wurde beim Bearbeiten eines Projekts beobachtet, wobei das SUT auch hier mit dem Statuscode `500` antwortete. Die Ursache dieses Fehlers konnte jedoch nicht genau ermittelt werden, da er nur sporadisch auftrat und sich nicht durch die Wiederholung einer bestimmten Testsequenz

reproduzieren ließ. Bei dem Fehler wurden auch unterschiedliche Log-Ausgaben produziert, wie:

```
"exception.class":"Rack::Timeout::RequestTimeoutException",
"exception.message":"Request ran for longer than 60000ms"
```

```
"exception.class":"Gitlab::Git::CommandError",
"exception.message":"14:last connection error: connection error:
desc = \"transport: Error while dialing dial unix
/var/opt/gitlab/gitaly/internal_sockets/ruby.1: connect:
connection refused\"."
```

Es lässt sich also annehmen, dass dieser Fehler auf Grund von Überlastung des SUT auftritt.

4.3 Diskussion

Im Folgenden sollen die erzielten Ergebnisse diskutiert werden. Dazu wird zunächst der Einfluss der zustandsabhängigen Properties auf das Resultat betrachtet. Im Abschnitt 3.2 wurden drei generische Zustände für Ressourcen vorgeschlagen: existierende, gelöschte und unbekannte Ressourcen. Beim Testen von GitLab ist jedoch aufgefallen, dass die davon abgeleiteten Properties nicht allgemeingültig sind. So setzt die Operation zum Wiederherstellen eines Projekts voraus, dass sich die Projekt-Ressource im gelöschten Zustand befindet. Das heißt, dass die Properties P7, die beim Aufruf einer gelöschten Ressource den Statuscode `404` (Not Found) oder `410` (Gone) erwartet, ebenso wie P8, die beim Aufruf einer bekannten Ressource einen Statuscode der Klasse `2XX` erwartet, nicht mehr zutreffen. Zusätzlich interferiert die anwendungsspezifische Geschäftslogik, insbesondere beim Bearbeiten von Ressourcen, mit den formulierten Bedingungen der zustandsabhängigen Properties. So darf, wie bereits beschrieben, der *Default Branch* nicht gelöscht werden. Ein weiteres Beispiel ist das Blockieren der Bearbeitung der Branch-Ressourcen, wenn das Projekt archiviert wurde. Die zustands-

abhängigen Properties verursachen also viele False-Positives, wenn diese als allgemeingültig betrachtet werden, und sind im generischen Kontext für das Testen von beliebigen Schnittstellen ungeeignet. Stattdessen müsste die Anwendbarkeit dieser Properties für die einzelnen Operationen in der Testkonfiguration anpassbar sein. Das grundsätzliche Speichern des Zustands einer Ressource hat sich jedoch bei der Erzielung einer breiten Abdeckung der Schnittstelle als Vorteil herausgestellt.

Als Nächstes sollen die im Rahmen der Tests gefundenen Fehler in den Kontext zu bestehenden Arbeiten gesetzt werden. So konnten durch REST-QA fünf unterschiedliche Fehler in GitLab gefunden werden. Im Vergleich dazu haben Atlidakis et al. 2019 acht Fehler für die Branch-Ressourcen und drei Fehler für die Project-Ressourcen gefunden. In der für die Tests genutzten Version von GitLab wurden jedoch bereits alle der von Atlidakis et al. 2019 auf *gitlab.com* gemeldeten Fehler korrigiert (vgl. Atlidakis 2018 a-g). Karlsson et al. 2020 nennen keine konkrete Anzahl der in GitLab gefundenen Fehler. Von den zwei in ihrer Arbeit referenzierten Fehlern wurde einer bereits behoben, während der zweite Fehler von REST-QA in F1 reproduziert werden konnte (vgl. Karlsson 2019 a-b). Die anderen gefundenen Fehler unterscheiden sich von solchen, die bereits durch ähnliche Arbeiten entdeckt wurden. Stattdessen konnte der bereits durch eine Entwicklerin manuell entdeckte Fehler F2 reproduziert werden. Mit den Fehlern F3 und F4 wurden sogar zwei vorher unbekannte Fehler in der Schnittstelle gefunden. Diese beiden Fehler wurden *gitlab.com* gemeldet und wurden als solche bestätigt (vgl. Kissmann 2021 a-b). Da sich der Fehler F5 nicht gezielt reproduzieren lässt, wird dieser im Rahmen dieser Arbeit nicht weiter betrachtet. REST-QA ist also in der Lage, in komplexen Schnittstellen Fehler zu produzieren und diese als solche zu erkennen. Alle gefundenen Fehler wurden dabei durch die

einzelne Property P4 – das SUT darf keine Antworten mit einem Statuscode der Klasse 5XX zurückgeben – erkannt. Da die ursprüngliche Online-Dokumentation nicht vollständig ist, wurden indirekt auch Fehler in dieser durch die Property P1 entdeckt, welche fordert, dass alle Antworten des SUT vollständig dokumentiert sind. Properties, die die Sicherheitsanforderungen überprüfen, konnten auf Grund des Sicherheitskonzepts von GitLab nicht validiert werden. Des Weiteren gilt es zu nennen, dass kein Fehler durch das Verwenden der zustandsabhängigen Properties erkannt wurde. Diese Aussage kann jedoch nicht als allgemeingültig betrachtet werden. Wie bereits beschrieben, ließen sich mit diesen Properties während der Entwicklung von REST-QA, komplexe Fehler finden, die zustandslose Properties nicht entdeckten können. Da letztendlich nur zwei Properties während der Analyse von GitLab Fehler finden konnten, kann die Aussage von Hughes 2013, dass das Verkomplizieren des Testmodells nicht nötig ist und schon mit wenigen Properties viele Fehler gefunden werden können, hiermit bestätigt werden.

5 FAZIT UND AUSBLICK

In dieser Arbeit wurde ein Konzept zur Validierung der Zuverlässigkeit und den Claim-basierten Berechtigungskonzepten von RESTful Webservices unter der Nutzung der *OpenAPI Specification* (OAS) entwickelt. Aufbauend auf diesem Konzept wurde daraufhin das Test-Tool REST-QA implementiert. Dabei wurden die Möglichkeiten und Limitationen der automatischen Analyse der OAS untersucht. Zusammenfassend wurde hierbei erkannt, dass es grundsätzlich keiner umfangreichen Anpassung der OAS bedarf, um die automatische Analyse zu ermöglichen. Essentiell ist dabei die Verwendung von Namenskonventionen für Referenzen. Trotzdem ist immer eine zusätzliche, manuelle Kontrolle und ggf. Anpassung der resultierenden Testkonfiguration durch die Testenden notwendig. Zudem besitzt REST-QA die Fähigkeit Claim-basierte Berechtigungskonzepte zu validieren. Um verschiedene Identitäten anzunehmen, unterstützt es die Authentifizierung über API-Schlüssel und den *Client-Credential-Flow* von OpenId Connect. Die Validierung der GitLab-Schnittstelle fokussiert sich jedoch auf die zustandsabhängigen Properties und die Analyse der Abhängigkeiten zwischen den Parametern. Die Ergebnisse konnten zeigen, dass sich das konzipierte Test-Tool sowie auch die zufällige Generierung von Testsequenzen eignet, um eine hohe Statuscodeabdeckung der einzelnen Operationen zu erreichen. Im Unterschied zu den Arbeiten von Atlidakis et al. 2019, Karlsson et al. 2020 und Viglianisi et al. 2020 werden bei der Generierung von Parameterbelegungen die Abhängigkeiten zwischen den Parametern betrachtet. So ermöglicht es die Testkonfiguration Eltern-Kind-Beziehungen zwischen den einzelnen Parametern einer Anfrage zu spezifizieren. Damit können auch bei voneinander abhängigen Parameter gezielt semantisch korrekte Anfrage an das SUT gesendet werden. Somit lässt sich das SUT effizient in Zustände versetzten, die die Vorbedingungen für

Operationen mit mehreren Abhängigkeiten zu anderen Ressourcen erfüllen. REST-QA generiert dabei nicht nur Anfragen für existierende Ressourcen, sondern wählt mit einer festen Gewichtung auch nicht existierende oder gelöschte Ressourcen aus. Des Weiteren wurden in dieser Arbeit zustandsabhängige Properties für existierende, unbekannte und gelöschte Ressourcen abgeleitet. Hier konnte bei der Validierung festgestellt werden, dass die Verwendung dieser drei generischen Zustände nicht allgemeingültig sein kann. Da das Verhalten jeder Operation von der Spezifikation der zustandsabhängigen Properties abweichen kann, wurde vorgeschlagen, diese Properties individuell konfigurierbar zu machen. Die konkrete Umsetzung sowie die Analyse der Ergebnisse dieser Anpassung müssen jedoch noch untersucht werden. Bei der Überprüfung der Schnittstelle von GitLab konnten mit den zustandsabhängigen Properties keine zusätzlichen Fehler gefunden werden. Dennoch ist nicht auszuschließen, dass diese Properties über die betrachtete Fallstudie hinaus das Potential haben, komplexe Fehler aufzudecken. Um den tatsächlichen Nutzen der zustandsabhängigen Properties bewerten zu können, sollten zukünftig weitergehende Tests durchgeführt und verschiedene RESTful Webservices untersucht werden. Insgesamt hat die Validierung die Aussage von Hughes 2013 bestätigt, dass nur wenige Properties ausreichen, um eine Vielzahl von Fehlern im SUT zu finden. Obwohl mit den zusätzlichen Properties keine weiteren Fehler gefunden wurden, gelang es REST-QA, zwei neue Fehler in der GitLab-Schnittstelle zu finden. Das entwickelte Test-Tool stellt damit ein weiteres Beispiel für den Nutzen des automatisierten Testens anhand der OAS dar.

In zukünftigen Arbeiten gilt es, diesen Ansatz an anderen RESTful Webservices zu testen. Bezogen auf REST-QA sollte zunächst ein Test für die gesamte GitLab-API ausgeführt werden. Das Test-Tool kann

um die Fähigkeit erweitert werden, gezielt fehlerhafte Anfrage zu generieren, wie es Viglianisi et al. 2020 vorschlagen. In diesem Zusammenhang kann es sinnvoll sein, bei der Generierung von invaliden Anfragen solche Parameter zu nutzen, die in anderen Operationen oder JSON-Schemata der Schnittstelle vorkommen. In der Validierung konnten auf Grund der Verwendung dieser Parameter die Fehler F2 und F3 gefunden werden. Auch wenn der Nutzen von zusätzlichen Properties in dieser Arbeit nicht nachgewiesen werden konnte, sind in Zukunft Erweiterungen für bestimmte Konzepte, wie die Paginierung, denkbar. Des Weiteren gilt es, die Unterstützung von REST-QA für andere Authentifizierungsmechanismen auszubauen. Anknüpfend an die Arbeit von Atlidakis et al. 2019 kann die Effektivität des Test-Tools untersucht werden, den Quelltext des SUT abzudecken und Fehler zu finden, in Abhängigkeit der bei der Testsequenz- und Wertegenerierung verwendeten Gewichtungen. Weitere Forschung kann auch im Bereich der automatisierten Analyse der OAS durchgeführt werden sowie der Erhöhung der Benutzbarkeit der Testkonfiguration, beispielsweise durch die Entwicklung einer domänenspezifischen Sprache zur Darstellungen der Abhängigkeiten zwischen den Ressourcen und Operationen.

A QUELLEN- UND LITERATURVERZEICHNIS

Atlidakis, V. (2018 a): 500 Internal Server Error when creating branch on double-deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50272 [29.05.2021].

Atlidakis, V. (2018 b): 500 Internal Server Error: Deleting branch of deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50678 [29.05.2021].

Atlidakis, V. (2018 c): 500 Internal Server Error: Creating a branch on a deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50950 [29.05.2021].

Atlidakis, V. (2018 d): 500 Internal Server Error: Getting branches of a deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50952 [29.05.2021].

Atlidakis, V. (2018 e): 500 Internal Server Error: Getting specific branch of a deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50953 [29.05.2021].

Atlidakis, V. (2018 f): 500 Internal Server Error: Editing specific branch of a deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50954 [29.05.2021].

Atlidakis, V. (2018 g): 500 Internal Server Error: (un)Protecting specific branch of a deleted project [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/50955 [29.05.2021].

Atlidakis, V., Godefroid, P. und Polishchuk, M. (2019): RESTler: Stateful REST API Fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, S. 748-758.

Ben-Kiki, O., Evans, C. und döt Net, I. (2009): YAML Ain't Markup Language (YAML™) Version 1.2 [online] https://yaml.org/spec/1.2/spec.html [27.03.2021].

Berners-Lee, T., Fielding, R. und Masinter, L. (2005): Uniform Resource Identifier (URI): Generic Syntax(RFC 3986). Network Working Group [online] https://tools.ietf.org/html/rfc3986 [06.01.2021].

Bray, T (2017): The JavaScript Object Notation (JSON) Data Interchange Format [online] https://tools.ietf.org/html/rfc8259 [12.02.2021].

Chappell, D. (2011): Claims-Based Identity for Windows [online] http://download.microsoft.com/download/7/D/0/7D0B5166-6A8A-418A-ADDD-95EE9B046994/Claims-Based%20Identity%20for%20Windows.pdf [22.04.2021].

Claessen, K. und Hughes, J. (2000): QuickCheck: a lightweight tool for random testing of Haskell programs, in: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00). Association for Computing Machinery, New York, NY, USA, S. 268–279.

Conklin, J. (1987): Hypertext: an introduction and Survey, in: computer, 20. Jg., Nr. 9, S. 17-41.

Docker Inc. (2021): Docker overview [online] https://docs.docker.com/get-started/overview/ [23.05.2021].

Duran, J. W. und Ntafos, S. C. (1984): An Evaluation of Random Testing, in: IEEE Transactions on Software Engineering, vol. SE-10, no. 4, S. 438-444.

Fielding, R. (2000): Architectural Styles and the Design of Network-based Software Architectures, Doktorarbeit, University of California, Irvine.

Fielding, R. und Reschke, J. (2014): Hypertext Transfer Protocol (HTTP/1.1): Authentication [online] https://tools.ietf.org/html/rfc7235 [16.02.2021].

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. und Berners-Lee, T. (1999): Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616 [online] https://www.rfc-editor.org/info/rfc2616 [09.11.2020].

Fink, G. und Bishop, M. (1997): Property-based testing: a new approach to testing for assurance, in: SIGSOFT Softw. Eng. Notes 22, 4, S. 74–80.

GitLab Inc. (o. J. a): About GitLab [online] https://about.gitlab.com/ [23.05.2021].

GitLab Inc. (o. J. b): GitLab Docker images [online] https://docs.gitlab.com/omnibus/docker/ [23.05.2021].

GitLab Inc. (o. J. c): API Docs [online] https://docs.gitlab.com/ee/api/ [23.05.2021].

Hamlet, R. (2002): Random Testing, in: Encyclopedia of Software Engineering, J.J. Marciniak (Ed.).

Hardt, D. (2021): The OAuth 2.0 Authorization Framework [online] https://tools.ietf.org/html/rfc6749 [22.04.2021].

Howden, W. (1980): Functional Program Testing, in: IEEE Transactions on Software Engineering, vol. 6, no. 02, S. 162-169.

Hughes, J. (2013): Race Conditions, Distribution, Interactions— Testing the Hard Stuff and Staying Sane [online] https://vimeo.com/68383317 [20.02.2021].

Jones, M. und Hardt, D. (2012): The OAuth 2.0 Authorization Framework: Bearer Token Usage [online] https://tools.ietf.org/html/rfc6750 [23.04.2021].

Karlsson, S. (2019 a): 500 Internal Server Error: per_page parameter with 0 [online] https://gitlab.com/gitlab-org/gitlab/-/issues/30114 [29.05.2021].

Karlsson, S. (2019 b): 500 Internal Server Error: large value for page parameter [online] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/64573 [29.05.2021].

Karlsson, S., Čaušević, A. und Sundmark, D. (2020): QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, S. 131-141.

Kissmann, B. (2021 a): 500 Internal Server Error: Create Project by User with default_branch [online] https://gitlab.com/gitlab-org/gitlab/-/issues/332197 [29.05.2021].

Kissmann, B. (2021 b): 500 Internal Server Error caused by missing input validation for tag length [online] https://gitlab.com/gitlab-org/gitlab/-/issues/332195 [29.05.2021].

Miller, D., Whitlock, J. Gardiner, M., Ralphson, M., Ratovsky, R. und Sarid, U. (2020): OpenAPI Specification, Version 3.0.3 [online] http://spec.openapis.org/oas/v3.0.3 [09.02.2021].

OpenAPI Initiative (2021): FAQ - OpenAPI Initiative [online] https://www.openapis.org/faq [16.02.2021].

Pautasso, C. (2014): RESTful Web Services: Principles, Patterns, Emerging Technologies, in: Bouguettaya A., Sheng Q., Daniel F. (eds) Web Services Foundations, New York, NY: Springer.

Peters, D. K. und Parnas L. (1994): Generating a Test Oracle from Program Documentation, in: Proceedings of the 1994 International Symposium on Software Testing and Analysis. ISSTA. ACM Press, S. 58-65.

Richardson, D. J., Aha, S. L. und O'Malley, T. O. (1992): Specification-based test oracles for reactive systems, in: Proceedings of the 14th international conference on Software engineering (ICSE '92). Association for Computing Machinery, New York, NY, USA, S. 105–118.

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B. und Mortimore, C. (2014): OpenID Connect Core 1.0 incorporating errata set 1 [online] https://openid.net/specs/openid-connect-core-1_0.html [22.04.2021].

Schermann, G., Cito, J. und Leitner, P. (2016): All the Services Large and Micro: Revisiting Industrial Practice in Services Computing, in: Norta A., Gaaloul W., Gangadharan G., Dam H. (eds) Service-Oriented Computing – ICSOC 2015 Workshops. ICSOC 2015. Lecture Notes in Computer Science, vol 9586. Springer, Berlin, Heidelberg.

Strnadl, C. (2020): OpenAPI - Sofware AG [online] https://techradar.softwareag.com/technology/openapi/ [16.02.2021].

Tilkov, S., Eigenbrodt, M., Schreier, S., und Wolf, O. (2015): REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web, Heidelberg: dpunkt. verlag.

Troschinetz, A. (2020): ci_default_git_depth allowed in create but will cause a 500 error [online] https://gitlab.com/gitlab-org/gitlab/-/issues/267392 [29.05.2021].

Tsoukalas, M. Z., Duran, J. W. und Ntafos, S. C. (1993): On some reliability estimation problems in random and partition testing, in: IEEE Trans. Softw. Eng. 19, 7, S. 687–697.

Utting, M. und Legeard, B. (2007): Practical model-based testing: a tools approach, San Francisco: Morgan Kaufmann.

Viglianisi, E., Dallago, M. und Ceccato, M. (2020): RESTTESTGEN: Automated Black-Box Testing of RESTful APIs, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, S. 142-152.

Wright, A. (2016): JSON Schema: A Media Type for Describing JSON Documents draft-wright-json-schema-00 [online] https://tools.ietf.org/html/draft-wright-json-schema-00 [09.02.2021].

B ABBILDUNGSVERZEICHNIS

Abbildung 2.1:	Ausstellung und Nutzung eines Tokens	19
Abbildung 3.1:	Kontext der Systemumgebung	26
Abbildung 3.2:	Datenfluss des Gesamtsystems	38
Abbildung 3.3:	White-Box-Ansicht REST-QA	39
Abbildung 3.4:	Prozesse der Test-Runner-Komponente	45
Abbildung 3.5:	White-Box-Ansicht Test Runner	46
Abbildung 3.6:	Prozesse zur Auswahl einer Testoperation	49
Abbildung 3.7:	Prozesse zur Generierung von Testwerten	53

C LISTING-VERZEICHNIS

Listing 2.1:	Repräsentation einer Ressource mit Ressourcenidentifikatoren in JSON-Notation	4
Listing 2.2:	HTTP-Antwort für GET api/players?page=1 (gekürzt)	12
Listing 2.3:	OpenAPI Specification in YAML-Repräsentation	15
Listing 2.4:	Operation mit Parametern und Anfrage-Body in YAML-Repräsentation	17
Listing 2.5:	Definition Sicherheitsschemaobjekt für OAuth2-Schema	18
Listing 2.6:	Verwendung eines Sicherheitsschemas mit OAuth2-Schema	18
Listing 3.1:	Repräsentation einer Spiel-Ressource mit vollständigen Ressourcenidentifikatoren	35
Listing 3.2:	Repräsentation einer Spiel-Ressource mit Ids	36
Listing 3.3:	Referenzen mit unterschiedlicher Semantik	37
Listing 3.4:	Testkonfiguration (verkürzt)	41
Listing 3.5:	Konfiguration der nutzbaren Identitäten	42
Listing 3.6:	Operation innerhalb der Testkonfiguration (verkürzt)	43
Listing 3.7:	PUT-HTTP-Anfrage (gekürzt)	47
Listing 3.8:	Ermittlung der Generatoroptionen (C#)	51

D TABELLENVERZEICHNIS

Tabelle 2.1: HTTP-Statuscodes 7

Tabelle 4.1: Liste der ausgewählten Operationen 55

Tabelle 4.2: Erzeugte Statuscodes für jede Operationen 58

E ABKÜRZUNGSVERZEICHNIS

BFS	Breadth-first search
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
MTTF	Mean Time to Failure
OAS	OpenAPI Specification
PBT	Property-Based Testing
REST	Representational State Transfer
RFC	Request for Comments
SSD	Solid State Drive
SUT	System Under Test
URI	Uniform Resource Identifiers
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

F ANHANG

Anhang 1: OpenAPI Specification GitLab

```yaml
openapi: "3.0.0"
info: { version: "13.11.2ce.0", title: "GitLab CE" }
security:
  - api_key: []
paths:
  /projects:
    get:
      parameters:
      - $ref: '#/components/parameters/page'
      - $ref: '#/components/parameters/per_page'
      - $ref: '#/components/parameters/archived'
      - $ref: '#/components/parameters/id_after'
      - $ref: '#/components/parameters/id_before'
      - name: last_activity_after
        in: query
        schema: { type: string, format: date-time }
      - name: last_activity_before
        in: query
        schema: { type: string, format: date-time }
      - $ref: '#/components/parameters/membership'
      - $ref: '#/components/parameters/min_access_level'
      - $ref: '#/components/parameters/Project_order_by'
      - $ref: '#/components/parameters/owned'
      - name: repository_checksum_failed
        in: query
        schema: { type: boolean }
      - name: repository_storage
        in: query
        schema: { type: string }
      - name: search_namespaces
        in: query
        schema: { type: boolean }
      - $ref: '#/components/parameters/search'
      - $ref: '#/components/parameters/simple'
      - $ref: '#/components/parameters/sort'
      - $ref: '#/components/parameters/starred'
      - $ref: '#/components/parameters/statistics'
      - name: topic
        in: query
        schema: { type: string }
      - $ref: '#/components/parameters/visibility'
      - name: wiki_checksum_failed
        in: query
        schema: { type: boolean }
      - $ref: '#/components/parameters/with_custom_attributes'
      - $ref: '#/components/parameters/Project_with_issues_enabled'
      - $ref: '#/components/parameters/Project_with_merge_requests_enabled'
      - $ref: '#/components/parameters/with_programming_language'
      responses:
        200: { $ref: '#/components/responses/Success_Project_List' }
        400: { $ref: '#/components/responses/400_BadRequest' }
    post:
      requestBody:
```

```
      content:
        application/json:
          schema: { $ref: '#/components/schemas/Project_Create' }
      required: true
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      400: { $ref: '#/components/responses/400_BadRequest' }
/projects/{id}:
  parameters:
  - $ref: '#/components/parameters/id'
  get:
    parameters:
    - name: license
      in: query
      schema: { type: boolean }
    - $ref: '#/components/parameters/statistics'
    - $ref: '#/components/parameters/with_custom_attributes'
    responses:
      200: { $ref: '#/components/responses/Success_Project' }
      400: { $ref: '#/components/responses/400_BadRequest' }
      404: { $ref: '#/components/responses/404_NotFound' }
  put:
    requestBody:
      content:
        application/json:
          schema: { $ref: '#/components/schemas/Project' }
      required: true
    responses:
      200: { $ref: '#/components/responses/Success_Project' }
      400: { $ref: '#/components/responses/400_BadRequest' }
      404: { $ref: '#/components/responses/404_NotFound' }
  delete:
    responses:
      202: { $ref: '#/components/responses/Success_Message_Response' }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/star:
  parameters:
  - $ref: '#/components/parameters/id'
  post:
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      304: { description: project is allready starred }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/unstar:
  parameters:
  - $ref: '#/components/parameters/id'
  post:
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      304: { description: project is not starred }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/starrers:
  parameters:
  - $ref: '#/components/parameters/id'
  get:
    parameters:
    - $ref: '#/components/parameters/page'
    - $ref: '#/components/parameters/per_page'
    - $ref: '#/components/parameters/search'
```

```
    responses:
      200:
        description: successful operation
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  starred_since: { type: string, format: date-time }
                  user: { $ref: '#/components/schemas/User' }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/archive:
  parameters:
  - $ref: '#/components/parameters/id'
  post:
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/unarchive:
  parameters:
  - $ref: '#/components/parameters/id'
  post:
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/restore:
  parameters:
  - $ref: '#/components/parameters/id'
  post:
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/user/{user_id}:
  parameters:
  - $ref: '#/components/parameters/user_id'
  post:
    requestBody:
      content:
        application/json:
          schema: { $ref: '#/components/schemas/Project_Create' }
      required: true
    responses:
      201: { $ref: '#/components/responses/Success_Project' }
      400: { $ref: '#/components/responses/400_BadRequest' }
      404: { $ref: '#/components/responses/404_NotFound' }
/users/{user_id}/projects:
  parameters:
  - $ref: '#/components/parameters/user_id'
  get:
    parameters:
    - $ref: '#/components/parameters/page'
    - $ref: '#/components/parameters/per_page'
    - $ref: '#/components/parameters/archived'
    - $ref: '#/components/parameters/id_after'
    - $ref: '#/components/parameters/id_before'
    - $ref: '#/components/parameters/membership'
    - $ref: '#/components/parameters/min_access_level'
```

```
    - $ref: '#/components/parameters/Project_order_by'
    - $ref: '#/components/parameters/owned'
    - $ref: '#/components/parameters/search'
    - $ref: '#/components/parameters/simple'
    - $ref: '#/components/parameters/sort'
    - $ref: '#/components/parameters/starred'
    - $ref: '#/components/parameters/statistics'
    - $ref: '#/components/parameters/visibility'
    - $ref: '#/components/parameters/with_custom_attributes'
    - $ref: '#/components/parameters/Project_with_issues_enabled'
    - $ref: '#/components/parameters/Project_with_merge_requests_enabled'
    - $ref: '#/components/parameters/with_programming_language'
    responses:
      200: { $ref: '#/components/responses/Success_Project_List' }
      400: { $ref: '#/components/responses/400_BadRequest' }
      404: { $ref: '#/components/responses/404_NotFound' }
/users/{user_id}/starred_projects:
  parameters:
  - $ref: '#/components/parameters/user_id'
  get:
    parameters:
    - $ref: '#/components/parameters/page'
    - $ref: '#/components/parameters/per_page'
    - $ref: '#/components/parameters/archived'
    - $ref: '#/components/parameters/membership'
    - $ref: '#/components/parameters/min_access_level'
    - $ref: '#/components/parameters/Project_order_by'
    - $ref: '#/components/parameters/owned'
    - $ref: '#/components/parameters/search'
    - $ref: '#/components/parameters/simple'
    - $ref: '#/components/parameters/sort'
    - $ref: '#/components/parameters/starred'
    - $ref: '#/components/parameters/statistics'
    - $ref: '#/components/parameters/visibility'
    - $ref: '#/components/parameters/with_custom_attributes'
    - $ref: '#/components/parameters/Project_with_issues_enabled'
    - $ref: '#/components/parameters/Project_with_merge_requests_enabled'
    responses:
      200: { $ref: '#/components/responses/Success_Project_List' }
      400: { $ref: '#/components/responses/400_BadRequest' }
      404: { $ref: '#/components/responses/404_NotFound' }
/projects/{id}/repository/branches:
  parameters:
  - $ref: '#/components/parameters/id'
  get:
    parameters:
    - $ref: '#/components/parameters/page'
    - $ref: '#/components/parameters/per_page'
    - $ref: '#/components/parameters/search'
    responses:
      200:
        description: successful operation
        content:
          application/json:
            schema:
              type: array
              items: { $ref: '#/components/schemas/Branch' }
      404: { $ref: '#/components/responses/404_NotFound' }
  post:
```

```yaml
      parameters:
      - name: branch
        in: query
        required: true
        schema: { type: string, minLength: 1, maxLength: 250 }
      - name: ref
        in: query
        required: true
        schema: { type: string, minLength: 1, maxLength: 250 }
      responses:
        201: { $ref: '#/components/responses/Success_Branch' }
        400: { $ref: '#/components/responses/400_BadRequest' }
        403: { $ref: '#/components/responses/403_Forbidden' }
        404: { $ref: '#/components/responses/404_NotFound' }
  /projects/{id}/repository/branches/{branch}:
    parameters:
    - $ref: '#/components/parameters/id'
    - name: branch
      in: path
      required: true
      schema: { type: string }
    get:
      responses:
        200: { $ref: '#/components/responses/Success_Branch' }
        404: { $ref: '#/components/responses/404_NotFound' }
    delete:
      responses:
        204: { description: successful operation }
        403: { $ref: '#/components/responses/403_Forbidden' }
        400: { $ref: '#/components/responses/400_BadRequest' }
        404: { $ref: '#/components/responses/404_NotFound' }
  /user:
    get:
      responses:
        200:
          description: successfull operation
          content:
            application/json:
              schema: { $ref: '#/components/schemas/User' }
components:
  schemas:
    Project_Base:
      type: object
      properties:
        id: { type: integer, readOnly: true }
        created_at: { type: string, format: date-time, readOnly: true }
        creator_id: { type: integer, readOnly: true }
        last_activity_at: { type: string, format: date-time, readOnly: true }
        marked_for_deletion_on: { type: string, format: date-time, readOnly: true }
        ssh_url_to_repo: { type: string, readOnly: true, nullable: true }
        http_url_to_repo: { type: string, format: uri, readOnly: true, nullable: true }
        web_url: { type: string, format: uri, readOnly: true }
        readme_url: { type: string, format: uri, readOnly: true, nullable: true }
        license_url: { type: string, format: uri, readOnly: true, nullable: true }
        license:
          type: object
          readOnly: true
          nullable: true
          properties:
```

```
      key: { type: string }
      name: { type: string }
      nickname: { type: string }
      html_url: { type: string, format: uri }
      source_url: { type: string, format: uri }
    additionalProperties: false
custom_attributes:
    type: array
    readOnly: true
    items: { $ref: '#/components/schemas/Key_Value_Pair' }
owner:
    allOf:
      - $ref: '#/components/schemas/User'
      - readOnly: true
can_create_merge_request_in: { type: boolean, readOnly: true }
archived: { type: boolean, readOnly: true }
empty_repo: { type: boolean, readOnly: true }
shared_with_groups:
    type: array
    readOnly: true
    items:
      type: object
      properties:
        group_id: { type: integer }
        group_name: { type: string }
        group_full_path: { type: string }
        group_access_level: { type: integer }
      additionalProperties: false
forks_count: { type: integer, readOnly: true }
star_count: { type: integer, readOnly: true }
open_issues_count: { type: integer, readOnly: true }
runners_token: { type: string, readOnly: true }
statistics:
    type: object
    readOnly: true
    properties:
      commit_count: { type: integer }
      storage_size: { type: integer }
      repository_size: { type: integer }
      wiki_size: { type: integer }
      lfs_objects_size: { type: integer }
      job_artifacts_size: { type: integer }
      packages_size: { type: integer }
      snippets_size: { type: integer }
    additionalProperties: false
container_registry_image_prefix: { type: string, readOnly: true }
permissions:
    type: object
    readOnly: true
    properties:
      project_access: { $ref: '#/components/schemas/Project_Access' }
      group_access: { $ref: '#/components/schemas/Project_Access' }
    additionalProperties: false
_links:
    type: object
    readOnly: true
    properties:
      self: { type: string }
      issues: { type: string }
```

```
            merge_requests: { type: string }
            repo_branches: { type: string }
            labels: { type: string }
            events: { type: string }
            members: { type: string }
          additionalProperties: false
        allow_merge_on_skipped_pipeline: { type: boolean, nullable: true }
        analytics_access_level: { $ref: '#/components/schemas/Access_Level' }
        approvals_before_merge: { type: integer }
        auto_cancel_pending_pipelines: { type: string, enum: [enabled, disabled] }
        auto_devops_deploy_strategy: { type: string, enum: [continuous, manual,
timed_incremental] }
        auto_devops_enabled: { type: boolean }
        autoclose_referenced_issues: { type: boolean }
        avatar_url: { type: string, format: uri, readOnly: true, nullable: true }
        build_coverage_regex: { type: string, minLength: 1, maxLength: 255, nullable:
true }
        build_git_strategy: { type: string, enum: [ fetch, clone ], default: fetch }
        build_timeout: { type: integer, minimum: 600, maximum: 2592000 }
        builds_access_level: {  $ref: '#/components/schemas/Access_Level' }
        ci_config_path: { type: string, nullable: true, maxLength: 255 }
        container_expiration_policy:
          allOf:
          - $ref: '#/components/schemas/Container_Expiration_Policy'
          - readOnly: true
        container_expiration_policy_attributes:
          allOf:
          - $ref: '#/components/schemas/Container_Expiration_Policy'
          - writeOnly: true
        container_registry_enabled: { type: boolean }
        default_branch: { type: string, nullable: true, maxLength: 250 }
        description: { type: string, nullable: true }
        emails_disabled: { type: boolean, nullable: true }
        external_authorization_classification_label: { type: string, maxLength: 255 }
        forking_access_level: { $ref: '#/components/schemas/Access_Level' }
        import_error: { type: object, readOnly: true, nullable: true }
        import_status: { type: string, readOnly: true }
        import_url: { type: string, writeOnly: true }
        issues_access_level: { $ref: '#/components/schemas/Access_Level' }
        issues_enabled: { type: boolean, readOnly: true }
        jobs_enabled: { type: boolean, readOnly: true }
        lfs_enabled: { type: boolean }
        merge_method: { type: string, enum: [merge, rebase_merge, ff] }
        merge_requests_access_level: { $ref: '#/components/schemas/Access_Level' }
        merge_requests_enabled: { type: boolean, readOnly: true }
        mirror_trigger_builds: { type: boolean }
        mirror: { type: boolean }
        name: { type: string, minLength: 1, maxLength: 255 }
        name_with_namespace: { type: string, readOnly: true }
        namespace:
          type: object
          readOnly: true
          properties:
            id: { type: integer }
            name: { type: string }
            path: { type: string }
            kind: { type: string }
            full_path: { type: string }
            parent_id: { type: integer, nullable: true }
```

```
        avatar_url: { type: string, format: uri, nullable: true }
        web_url: { type: string, format: uri }
      additionalProperties: false
    namespace_id: { type: integer, writeOnly: true }
    operations_access_level: { $ref: '#/components/schemas/Access_Level' }
    only_allow_merge_if_all_discussions_are_resolved: { type: boolean }
    only_allow_merge_if_pipeline_succeeds: { type: boolean }
    packages_enabled: { type: boolean }
    pages_access_level: { $ref: '#/components/schemas/Public_Access_Level' }
    requirements_access_level: { $ref: '#/components/schemas/Public_Access_Level' }
    path: { type: string, minLength: 1, maxLength: 255 }
    path_with_namespace: { type: string, readOnly: true }
    public_builds: { type: boolean, writeOnly: true }
    public_jobs: { type: boolean, readOnly: true }
    remove_source_branch_after_merge: { type: boolean }
    printing_merge_request_link_enabled: { type: boolean }
    repository_access_level: { $ref: '#/components/schemas/Access_Level' }
    repository_storage: { type: string }
    request_access_enabled: { type: boolean }
    resolve_outdated_diff_discussions: { type: boolean }
    shared_runners_enabled: { type: boolean }
    show_default_award_emojis: { type: boolean }
    snippets_access_level: { $ref: '#/components/schemas/Access_Level' }
    snippets_enabled: { type: boolean, readOnly: true }
    suggestion_commit_message: { type: string, nullable: true, maxLength: 255 }
    tag_list:
      type: array
      items: { type: string, maxLength: 255 }
    visibility: { $ref: '#/components/schemas/Visibility' }
    wiki_access_level: { $ref: '#/components/schemas/Access_Level' }
    wiki_enabled: { type: boolean, readOnly: true }
    issues_template: { type: string, nullable: true, maxLength: 255 }
    merge_requests_template: { type: string, nullable: true, maxLength: 255 }
  additionalProperties: false
Project:
  allOf:
  - type: object
    properties:
      ci_default_git_depth: { type: integer, minimum: 0, maximum: 1000 }
      ci_forward_deployment_enabled: { type: boolean }
      mirror_user_id: { type: integer }
      restrict_user_defined_variables: { type: boolean }
      service_desk_enabled: { type: boolean }
      service_desk_address: {type: string, nullable: true }
    additionalProperties: false
  - $ref: '#/components/schemas/Project_Base'
Project_Create:
  allOf:
  - $ref: '#/components/schemas/Project_Base'
  - type: object
    required:
    - name
    properties:
      group_with_project_templates_id: { type: integer }
      initialize_with_readme: { type: boolean }
      name: { type: string }
      template_name: { type: string }
      template_project_id: { type: integer }
      use_custom_template: { type: boolean }
```

```
    additionalProperties: false
  User:
    type: object
    properties:
      id: { type: integer }
      username: { type: string }
      name: { type: string }
      state: { type: string }
      avatar_url: { type: string, format: uri }
      web_url: { type: string, format: uri }
      created_at: { type: string, format: date-time, nullable: true }
      bio: { type: string, nullable: true }
      bio_html: { type: string, nullable: true }
      bot: { type: boolean, nullable: true }
      work_information: { type: string, nullable: true }
      followers: { type: integer, readOnly: true }
      following: { type: integer, readOnly: true }
      location: { type: string, nullable: true }
      public_email: { type: string, nullable: true }
      skype: { type: string, nullable: true }
      linkedin: { type: string, nullable: true }
      twitter: { type: string, nullable: true }
      website_url: { type: string, nullable: true }
      organization: { type: string, nullable: true }
      job_title: { type: string, nullable: true }
      last_sign_in_at: { type: string, format: date-time, nullable: true }
      confirmed_at: { type: string, format: date-time, nullable: true }
      theme_id: { type: integer, nullable: true }
      last_activity_on: { type: string, format: date, nullable: true }
      email: { type: string, format: email, nullable: true }
      color_scheme_id: { type: integer, nullable: true }
      projects_limit: { type: integer, nullable: true }
      current_sign_in_at: { type: string, format: date-time, nullable: true }
      is_admin: { type: boolean, nullable: true }
      note: { type: string, nullable: true }
      identities:
        type: array
        items:
          type: object
          properties:
            provider: { type: string }
            extern_uid: { type: string }
        nullable: true
      can_create_group: { type: boolean, nullable: true }
      can_create_project: { type: boolean, nullable: true }
      two_factor_enabled: { type: boolean, nullable: true }
      external: { type: boolean, nullable: true }
      private_profile: { type: boolean, nullable: true }
      commit_email: { type: string, format: email, nullable: true }
      current_sign_in_ip: { type: string, nullable: true }
      last_sign_in_ip: { type: string, nullable: true }
      plan: { type: string, nullable: true }
      trial: { type: boolean, nullable: true }
      sign_in_count: { type: integer, nullable: true }
      custom_attributes:
        type: array
        readOnly: true
        items: { $ref: '#/components/schemas/Key_Value_Pair' }
    additionalProperties: false
```

```yaml
Branch:
  type: object
  properties:
    name: { type: string }
    merged: { type: boolean }
    protected: { type: boolean }
    default: { type: boolean }
    developers_can_push: { type: boolean }
    developers_can_merge: { type: boolean }
    can_push: { type: boolean }
    web_url: { type: string, format: uri }
    commit: { $ref: '#/components/schemas/Commit' }
  additionalProperties: false
Commit:
  type: object
  properties:
    created_at: { type: string, format: date-time }
    authored_date: { type: string, format: date-time }
    author_email: { type: string, format: email }
    author_name: { type: string }
    author_date: { type: string, format: date-time }
    committed_date: { type: string, format: date-time }
    committer_email: { type: string, format: email }
    committer_name: { type: string }
    id: { type: string }
    short_id: { type: string }
    title: { type: string }
    message: { type: string }
    parent_ids:
      type: array
      nullable: true
      items: { type: string }
    web_url: { type: string, format: uri }
  additionalProperties: false
Container_Expiration_Policy:
  type: object
  properties:
    cadence: { type: string, enum: [ 1d, 7d, 14d, 1month, 3month ] }
    enabled: { type: boolean }
    keep_n: { type: integer, enum: [ 1, 5, 10, 25, 50, 100 ] }
    older_than: { type: string, enum: [ 7d, 14d, 30d, 90d ] }
    name_regex: { type: string, minLength: 1, maxLength: 255 }
    name_regex_keep: { type: string, minLength: 1, maxLength: 255, nullable: true }
    next_run_at: { type: string, format: date-time, readOnly: true }
  additionalProperties: false
Key_Value_Pair:
  type: object
  properties:
    key: { type: string, maxLength: 255 }
    value: { type: string, maxLength: 255 }
  additionalProperties: false
Access_Level: { type: string, enum: [disabled, private, enabled] }
Public_Access_Level: { type: string, enum: [public, disabled, private, enabled] }
Visibility: { type: string, enum: [public, internal, private] }
Message_Response:
  type: object
  properties:
    message:
      anyOf: [ { type: object }, { type: string } ]
```

```yaml
      error:
        anyOf: [ { type: object }, { type: string } ]
  Project_Access:
    type: object
    nullable: true
    properties:
      access_level: { type: integer }
      notification_level: { type: integer }
    additionalProperties: false
parameters:
  id:
    name: id
    in: path
    required: true
    schema: { type: integer }
  user_id:
    name: user_id
    in: path
    required: true
    schema: { type: integer }
  archived:
    name: archived
    in: query
    schema: { type: boolean }
  id_after:
    name: id_after
    in: query
    schema: { type: integer }
  id_before:
    name: id_before
    in: query
    schema: { type: integer }
  membership:
    name: membership
    in: query
    schema: { type: boolean }
  min_access_level:
    name: min_access_level
    in: query
    schema: { type: integer, enum: [10, 20, 30, 40, 50] }
  owned:
    name: owned
    in: query
    schema: { type: boolean }
  page:
    name: page
    in: query
    schema: { type: integer, minimum: 1, maximum: 500, default: 1 }
  per_page:
    name: per_page
    in: query
    schema: { type: integer, minimum: 1, maximum: 100, default: 20 }
  search:
    name: search
    in: query
    schema: { type: string }
  simple:
    name: simple
    in: query
```

```
      schema: { type: boolean }
    sort:
      name: sort
      in: query
      schema: { type: string, enum: [asc, desc], default: desc }
    starred:
      name: starred
      in: query
      schema: { type: boolean }
    statistics:
      name: statistics
      in: query
      schema: { type: boolean }
    visibility:
      name: visibility
      in: query
      schema: { $ref: '#/components/schemas/Visibility' }
    with_custom_attributes:
      name: with_custom_attributes
      in: query
      schema: { type: boolean }
    with_programming_language:
      name: with_programming_language
      in: query
      schema: { type: string }
    Project_order_by:
      name: order_by
      in: query
      schema:  { type: string, enum: [id, name, path, created_at, updated_at,
last_activity_at, repository_size, storage_size, packages_size, wiki_size] }
    Project_with_issues_enabled:
      name: with_issues_enabled
      in: query
      schema: { type: boolean }
    Project_with_merge_requests_enabled:
      name: with_merge_requests_enabled
      in: query
      schema: { type: boolean }
  responses:
    Success_Project:
      description: Successful operation
      content:
        application/json:
          schema: { $ref: '#/components/schemas/Project' }
    Success_Project_List:
      description: Successful operation
      content:
        application/json:
          schema:
            type: array
            items: { $ref: '#/components/schemas/Project' }
    Success_Branch:
      description: Successful operation
      content:
        application/json:
          schema: { $ref: '#/components/schemas/Branch' }
    Success_Message_Response:
      description: Successful operation
      content:
```

```
      application/json:
        schema: { $ref: '#/components/schemas/Message_Response' }
  400_BadRequest:
    description: Bad request
    content:
      application/json:
        schema: { $ref: '#/components/schemas/Message_Response' }
  403_Forbidden:
    description: Forbidden
    content:
      application/json:
        schema: { $ref: '#/components/schemas/Message_Response' }
  404_NotFound:
    description: Object not found
    content:
      application/json:
        schema: { $ref: '#/components/schemas/Message_Response' }
securitySchemes:
  api_key:
    type: apiKey
    name: Private-Token
    in: header
```