Inhaltsverzeichnis

1 Einführung in die Thematik **2**

1.1 HTTP 2

1.2 REST 4

1.3 Spezifikationsformate 5

1.3.1 JSON Schema 6

1.3.2 OpenAPI 6

1.4 Testing von REST APIs 7

1.4.1 Funktionstests 7

1.4.2 Last- und Stresstests 8

1.4.3 Sicherheitstests 8

1.5 Verwandte Arbeiten und Systeme 8

1 Einführung in die Thematik

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

1.1 HTTP

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

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

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

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

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

Body: Enhält die Nutzdaten der Nachricht.

Das HTTP Protokoll erlaubt die folgenden Operationen bzw. Methoden:

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

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

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

DELETE: Die angegebene Ressource löschen.

PUT: Die angegebene Ressource mit neuen Daten ersetzen.

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

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

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

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

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

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

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

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

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

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

1.2 REST

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

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

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

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

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

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

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

1.3 Spezifikationsformate

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

1.3.1 JSON Schema

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

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

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

1.3.2 OpenAPI

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

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

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

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

1.4 Testing von REST APIs

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

1.4.1 Funktionstests

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

gendsten Form sollte ein korrekter Response-Body die Darstellung der angeforderten Ressource unter Verwendung des angeforderten Medientyps oder mindestens einer gültigen HTTP-Fehlercodemeldung enthalten. Funktionstests sollten nicht nur den regulären Testfall einbeziehen, sondern auch Szenarien von Errata- und Edge-Cases in das Testverfahren implementieren. Ein Beispiel hierfür ist eine Anfrage mit falschen Attributen oder Attributwerten, auf die die API mit einem korrekten Fehlercode antworten sollte.

1.4.2 Last- und Stresstests

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

1.4.3 Sicherheitstests

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

1.5 Verwandte Arbeiten und Systeme

Literatur

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

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

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

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

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

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

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

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

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

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