Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Fachstudie Nr. 197

Musterlösungen und Best Practices für das Design und die Realisierung von REST Schnittstellen

Marc Schmid, Tom Rohloff, Philipp Duwe

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. Frank Leymann
Betreuer: Dipl.-Inf. Florian Haupt
begonnen am: 07.04.2014
beendet am: 07.10.2014

CR-Klassifikation: D.2.1, C.2.4, H.5.4

Zusammenfassung

Durch die wenigen und abstrakten Grundregeln für REST-Schnittstellen gibt es eine Vielzahl von Meinungen und möglichen Implementierungen, welche der Definition für REST-Schnittstellen folgen wollen. Diese Meinungen sind teilweise widersprüchlich zueinander und befolgen nicht immer die Grundprinzipien die notwendig sind, um eine Schnittstelle als RESTful zu bezeichnen. In dieser Ausarbeitung werden die gängigsten Musterlösungen und Best Practices für das Design und die Realisierung von REST-Schnittstellen vorgestellt und ihre Vor- und Nachteile diskutiert, sowie anhand der Schnittstelle der Plattform GitHub praktisch dargelegt.

Inhaltsverzeichnis

Tabellenverzeichnis

1 Einleitung

Durch die Definition des Representational State Transfers (REST) wurde von Roy Fielding [22] eine Paradigma erschaffen, welches Grundregeln für eine Art von Schnittstellen definiert, welche im World Wide Web kommunizieren. Da diese Definition jedoch nur einige Grundregeln beinhaltet, wurden von vielen Autoren und Softwareentwicklern verschiedene Meinungen und Methoden entwickelt, wie diese Grundregeln in konkrete Schnittstellen umzusetzen sind. Diese Meinungen und Methoden sind teilweise doch sehr verschieden und bieten mehrere Ansätze, um Schnittstellen so zu konstruieren, damit diese als RESTful bezeichnet werden können. Dabei können sich diese Ansätze darin unterscheiden, dass sie nur softwaretechnische Aspekte besitzen, wie die Benennung von Ressourcen, bis hin zu Implementierungsdetails, welche sogar gegen das verwendete Protokoll (etwa HTTP) verstoßen. Um eine Übersicht über die häufigsten Best Practices und Musterlösungen beim Design und Entwurf von REST-Schnittstellen zu bieten und diese zukünftigen REST-Schnittstellenentwicklern als Möglichkeiten anzubieten, wurden in dieser Fachstudie die gängigsten Musterlösungen und Best Practices beim Design und Entwurf von REST-Schnittstellen zusammengeführt, die von ihren jeweiligen Autoren aufgeführten, Vor- und Nachteile aufgelistet, sowie eine Bewertung der jeweiligen Lösungen dieser Fachstudie beigefügt.

Nach dem Einführungsabschnitt wird in Abschnitt 2 die Methodik bei der Recherche und der Erstellung dieser Fachstudie erläutert. In Abschnitt 3 werden die Grundbegriffe und Technologien erläutert, welche für das Verständnis von Representational State Transfers (REST) nötig sind. Abschnitt 4 enthält die Vergleiche der Designentscheidungen für REST-Schnittstellen. In diesem Abschnitt werden die empfohlenen Musterlösungen und Best Practices vorgestellt und miteinander verglichen, in dem ihre Vor- und Nachteile aufgelistet werden. Ebenfalls wird eine Bewertung der Lösungen der jeweiligen Autoren dieser Fachstudie zu den vorgestellten Musterlösungen aufgeführt. Um ein praktisches Beispiel für eine RESTful Schnittstelle zu haben, wird in Abschnitt 5 die GitHub-API mit den vorher vorgestellten Musterlösungen und Best Practices verglichen. Der letzte Abschnitt enthält das Fazit dieser Fachstudie.

2 Methodik

Dieses Kapitel deckt die Methodik ab, die angewandt wurde, um diese Fachstudie zu erstellen. Ferner wird auf den Unterschied zwischen REST als Architekturstil und HTTP als eine Implementierung hiervon eingegangen.

Eines der Hauptprobleme bei der Recherche war das Gewinnen eines Überblicks über weitere Standpunkte und Ideen, welche teils erheblich von Roy Fieldings Ansichten abweichen. Insbesondere der Spagat zwischen dem Befolgen dessen, was das Protokoll vorschreibt, und den mit der Zeit etablierten Mustern, welche oft erheblich davon abweichen.

Viele Ideen wurden und werden über Foren oder Blogs kommuniziert. Um diese Entwicklungen abseits der Standardliteratur ebenfalls zu berücksichtigen und sich so ein genaueres Bild über die aktuelle Lage zu verschaffen, haben wir Suchmaschinen benutzt. Um die Ergebnisse auf ein bewältigbares Maß einzugrenzen, haben wir mit der Suchmaschine Google (`https://www.google.com/ncr`) folgende Suchen durchgeführt:

- `rest api design`
- `rest api best practice`
- `restful api design`
- `restful api best practice`
- `restful api versioning`
- `rest versioning`
- `rest best practice`
- `rest ressource`
- `rest api different languages`

Ferner haben wir bei jeder Suche die ersten 30 Ergebnisse (die ersten 3 Ergebnisseiten) angesehen und aus diesen nur jene Ergebnisse verwendet, welche sich mit den abstrakteren Prinzipien von REST beschäftigen und weniger mit konkreten Implementierungen in Programmiersprachen. Dieses Filtern wurde über den Vorschautext (Beschreibungstext) und den Titel des jeweiligen Suchergebnisses vorgenommen. Wurden innerhalb des dann gefundenen Suchergebnisses Verweise auf andere Quellen aufgeführt, so sind wir diesen weiter gefolgt.

Wir möchten für diese Ausarbeitung anmerken, dass der Representational State Transfer (REST) genau genommen eine Abstraktion der Architektur des World Wide Web ist, wodurch dieses Paradigma für verschiedene Implementierungen und Protokolle gültig und möglich ist. Da allerdings ein großer Teil des World Wide Web durch das Hypertext Transfer Protocol (HTTP) [**?**] realisiert wird und alle Autoren, auf denen diese Fachstudie beruht, in ihren Werken RESTful APIs via HTTP beschreiben, werden auch wir uns in dieser Fachstudie auf HTTP als reale Implementierung der abstrakten Architektur beschränken. Dennoch muss im Hinterkopf behalten werden, dass REST einen Architekturstil beschreibt, keine Implementierung selbst.

3 Grundlagen

In Kapitel 3 werden die nötigen technischen Grundlagen behandelt, welche für das Verständnis von REST APIs nötig sind und ein besseres Verständnis darüber geben sollen, wie REST beschrieben wird. Es werden zuerst die technische Begriffe für REST APIs behandelt, gefolgt von den Prinzipien, welche eine REST API erfüllen muss, um als RESTfull zu gelten. Am Ende des Kapitels wird kurz auf "Hypermedia as the Engine of Application State" (HATEOAS) [1] und das Richardson Maturity Model [29] eingegangen.

3.1 Begriffserklärung der Technologien

In diesem Unterkapitel werden die technischen Begriffe kurz erklärt, welche für das Verständnis der Designentscheidungen von REST APIs auf HTTP-Basis notwendig sind. In erster Linie sind dies Begriffe des HTTP, auch wenn diese Begriffe allgemeiner gehandhabt werden können.

3.1.1 Was ist REST?

REST bezeichnet zunächst einen Architekturstil für die Kommunikation zwischen Maschinen, der ursprünglich Dissertation von Roy Fielding [22] definiert wurde. Das Programmierparadigma der Representational State Transfers konzentriert sich eher auf die Rollenverteilung und das Rollenverständnis der Kommunikationspartner und der Inhalte, als auf genaue Implementierungen wie etwa eine Syntax. Ein angebotener Dienst, der "RESTful" ist - also dem Paradigma folgt - erfüllt bestimmte Prinzipien, die in Abschnitt 3.3 näher erläutert werden. REST steht eine Abstraktionsebene über der konkreten Implementierung. Eine solche Umsetzung ist zum Beispiel das weit verbreitete HTTP, welches im folgenden Abschnitt beschrieben wird.

3.1.2 HTTP

Das Hypertext Transfer Protocol [**?**] bezeichnet ein Protokoll für den Austausch von Daten in Netzwerken. Im OSI-Schichtenmodell für Netzwerke, welches eine Einteilung der Netzwerkprotokolle hinsichtlich ihrer Nähe zum Benutzer beziehungsweise zur Hardware vornimmt, liegt HTTP über der Transportschicht 4 auf den Schichten 5-7. HTTP ist dem Benutzer also vergleichsweise nahe - im Gegensatz zu den darunterliegenden Protokollen wie TCP oder IP.

Im nun folgenden Absatz sollen die Grundprinzipien einer HTTP-Kommunikation erläutert werden. Grundsätzlich gibt es eine Unterscheidung der Kommunikationspartner in Client und Server. Der Client stellt eine Anfrage (Request) an den Server. Dieser antwortet anschließend (Response). Ein Request hat die folgende grundsätzliche Form.

- Eine Anfragezeile, welche die URL (Uniform Resource Locator) [2] der gewünschten Ressource enthält. Zusätzlich steht hier, welche Version von HTTP genutzt und welche Operation ausgeführt wird
- Anfrage-Header-Felder: Hier können verschiedene Anfrageheader untergebracht werden. Dabei wird pro Header-Feld eine Zeile genutzt. Header-Felder können zum Beispiel die gewünschte Sprache (`Accept-Language`), die Version (`Accept` oder Inhaltsformat (Content-Type) sein.
- eine Leerzeile
- optionaler Inhalt (Message Body)

Die Anfragezeile sowie die Header-Felder müssen mit den Sonderzeichen <CR> (carriage return, Zeilenumbruch) und <LF> (line feed, Zeilenvorschub) abgeschlossen werden. Die Leerzeile darf ausschließlich diese beiden Zeichen enthalten (und keine Whitespaces wie Leerzeichen oder Tabulatoren).

Eine korrekte Anfrage sieht wie folgt aus:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Hierbei wird auf dem Server, welcher den Domainnamen `www.example.com` hat, das Ziel `index.html` über das Protokoll `HTTP 1.1` mit der Operation `GET` angefragt. Der vollständige Pfad (URL) sieht also so aus: `http://www.example.com/index.html`

Eine Antwort hingegen folgt diesem Schema:

- Eine Statuszeile, welche den entsprechenden HTTP-Statuscode, sowie dessen Kurzbeschreibung enthält
- Antwort-Header-Felder: Hier kann etwa stehen, in welchem Format der Inhalt zurückgegeben wird.
- eine Leerzeile
- optionaler Inhalt

Ähnlich wie bei der Anfrage müssen die Statuszeile und alle Header-Felder mit <CR> und <LF> abgeschlossen werden. Die Leerzeile darf ebenso nur aus diesen beiden Zeichen bestehen.

Die Antwort des angefragten Servers auf den obenstehenden Request könnte somit wie folgt aussehen:

```
HTTP/1.1 200 OK
Date: Sat, 14 Feb 2009 00:31:30 CET
Content-Type: text/html; charset=UTF-8
Content-Length: 118

<html>
<head>
<title>Cato Censorius</title>
</head>
<body>
Ceterum censeo Carthaginem esse delendam.
</body>
</html>
```

HTTP-Statuscodes

HTTP-Statuscodes werden vom Server an den Client gesendet, nachdem dieser seine Anfrage gestellt hat. So kann der Client vom Server erfahren, ob seine Anfrage angekommen ist und ob diese bearbeitet werden kann, oder wie weit die Anfrage schon bearbeitet wurde. So kann dem Client auch vermittelt werden, ob die Anfrage fehlerhaft war oder ob die erwünschten Ressourcen nicht mehr verfügbar sind.

Dadurch sind HTTP-Statuscodes für REST-APIs außerordentlich wichtig, da durch ihren Inhalt der Client direkt reagieren und seine Anfrage dem Statuscode entsprechend anpassen kann.

Die nachfolgende Tabelle soll einen kurzen Überblick über einige HTTP-Statuscodes bieten:

Code	Text	Bedeutung
200	OK	Die Anfrage wurde erfolgreich bearbeitet und ihr Ergebnis ist in der Antwort enthalten.
201	Created	Die Anfrage wurde erfolgreich bearbeitet und die angeforderte Ressource wurde erstellt.
204	No Content	Die Anfrage wurde erfolgreich durchgeführt, die Antwort enthält jedoch keine Daten.
304	Not Modified	Der Inhalt der Ressource hat sich seit der letzten Abfrage durch den Client nicht verändert und wird deshalb nicht übertragen.
400	Bad Request	Die Anfrage war fehlerhaft.
401	Unauthorized	Anfrage kann nicht ohne erfolgreiche Authentifizierung bearbeitet werden.
403	Forbidden	Die Anfrage wurde mangels Berechtigung nicht durchgeführt.
404	Not Found	Die angeforderte Ressource wurde nicht gefunden.
405	Method Not Allowed	Die Anfrage darf nur mit anderen HTTP-Methoden angefragt werden.
415	Unsupported Media Type	Der Inhalt der Anfrage wurde mit einem nicht unterstützen Medientyp übermittelt.
500	Internal Server Error	Ein nicht erwarteter Fehler ist serverseitig aufgetreten.

Tabelle 1: HTTP-Statuscodes

3.1.3 Ressourcen

Eine Ressource ist eine nicht sichtbare, durch einen gemeinsamen Identifikator zusammengehaltene Menge von Repräsentationen, wobei eine einzelne Ressource mehrere unterschiedliche Repräsentationen haben kann, die es erlauben die Ressource in mehreren vordefinierten Formaten anzubieten. Eine Ressource könnte im übertragenen Sinne mit den Objekten der objektorientierten Programmierung verglichen werden. Da die Ressourcen einer der Mittelpunkte eines REST-Architekturentwurfs sind, werden Architekturen, die im REST-Stil implementiert werden, auch Resource Oriented Architecture(ROA) genannt.

Nach Tilkov [41] werden Ressourcen in folgende verschiedene Kategorien unterteilt:

Primärressourcen sind die wichtigsten Entitäten eines klassischen Entwurfs, da sie den Kern einer Anwendung bilden. Ihre Implementierung ist für den Nutzer komplett transparent und lässt keine Rückschlüsse auf die Implementierung der gedachten Anwendung zu. Beispiele für Ressourcen in einer gedachten URI:

```
http://example.com/user/1234
http://example.com/account/exampleaccount
```

Subressourcen sind Teile einer Ressource, zum Beispiel ein einzelnes Rezept in einem Kochbuch, eine Lieferadresse einer Bestellung oder die nähere Beschreibung einer Person einer Datenbank von Mitarbeitern. Hier wird sichtbar, dass es ausschließlich eine Entscheidung der Entwickler ist, ob ein gewisses Modellelement auf eine eigene Subressource abgebildet wird, oder ob in eine schon bestehende Ressource mehrere Repräsentationen eingebettet werden.

Listenressourcen sind in den meisten Fällen eine Zusammenfassung von verschiedenen Primärressourcen, beispielweise die Liste der Rezepte oder die Liste der Mitarbeiter. Die Tatsache, dass Listenressourcen auch Ressourcen sind, bedeutet, dass diese eindeutig identifiziert werden und voraussichtlich mehr als eine Repräsentation haben. Als Antwort auf die vorigen Anfragen per GET erhält man eine Repräsentation der Liste, per POST kann eine neue Primärressource als Element der Liste angelegt werden.

Filter sind auf Listen anwendbar, um die Auswahl der Ergebnisse auf gewisse Kriterien einzuschränken. Somit können beispielsweise die Ergebnisse der Rezepte auf vegetarische Rezepte eingegrenzt werden, wobei diese dann auch Kandidaten für Ressourcen sind.

Projektionen bieten die Möglichkeit, nur einen Teil gewisser verfügbarer Information einer Primärressource oder nur einige ausgewählte Elemente einer Liste abzufragen. So kann also eine Menge an Informationen in eine sinnvolle Untermenge unterteilt werden und somit die zu übertragene Datenmenge verringert werden.

Aggregationen erlauben es - im Gegensatz zu einer Projektion - die Information mehrerer Listen- oder Primärressourcen zusammenzufassen und somit die Anzahl der Server/Client-Interaktionen zu begrenzen.

3.1.4 Operationen

Roy Fielding legt in seiner Dissertation über REST fest, dass es für REST-Architekturen eine definierte, begrenzte Menge von Operationen geben muss. Allerdings spezifiziert er die Operationen nicht konkret. Es ist allerdings empfehlenswert, dass die Operationen bestimmte Eigenschaften erfüllen müssen: idempotent und/oder sicher. Diese Begriffe werden nun erklärt.

Idempotente Operationen: Eine Operation wird als idempotent [3] gesehen, wenn die mehrmalige Ausführung einer gleichen Anfrage den Zustand des Server nicht weiter verändert. Idempotente Operationen sind vergleichbar mit Setter-Methoden in Programmiersprachen: Auch nach mehrmaligem Aufruf mit dem gleichen Wert enthält die Variable den gleichen Wert, wie nach dem erstem Aufruf mit diesem Wert.

Sichere Operationen (safety): Eine Operation gilt als sicher, wenn ihre Ausführung den Zustand des Servers nicht verändert. Dies bedeutet, dass sichere Operationen keine Nebeneffekte haben dürfen, um unfreiwillige Änderungen am Serverzustand zu verhindern. Deshalb sind sichere Operationen meistens dafür gedacht um Informationen vom Server zu erhalten. Falls ein Server Anfragen mitloggt oder andere statischen Änderungen bei der Anfrage durchführt, werden diese Änderungen trotzdem durchgeführt, was allerdings der Definition von sicheren Operationen nicht zuwider handelt. Eine sichere Operation ist auch immer idempotent, da ihr mehrmaliger Aufruf zum selben Ergebnis auf dem Server führt.

HTTP-Operationen: Wenn eine REST-Architektur über HTTP realisiert wird, werden für die Operationen die HTTP-Verben verwendet. Im Nachfolgenden werden die vier HTTP-Verben GET, POST, PUT und DELETE kurz erläutert. Die Verben HEAD, OPTIONS, TRACE und CONNECT werden der Vollständigkeit halber noch kurz behandelt.

GET: Die GET-Operation ist die wichtigste Operation in einer HTTP-basierten REST-Architektur. Es wird angenommen, dass mehr als 95% aller Interaktionen im World Wide Web lesende Operationen sind [41]. GET dient zur Abfrage einer Entität/Repräsentation. Da es eine lesende Operation ist, darf GET niemals etwas auf dem Server updaten, erstellen oder löschen. Folglich müssen GET-Implementierungen sicher und idempotent sein.
Eine GET-Anfrage sieht in ihrer Grundstrucktur wie folgt aus:

```
GET http://example.org/rest-fs/texte/275
```

Auch besteht die Möglichkeit, weitere Header in der Anfrage anzugeben. Dies kann beispielsweise über das `Accept` Feld realisiert werden. Über das `Accept` Feld teilt der Client dem Server mit, welche Datentypen er verarbeiten kann und schränkt damit die Antwort des Servers auf die Anfrage ein. Die meisten Implementierungen nehmen an, dass sobald kein Accept Feld angegeben ist, der Client alle Repräsentationen der Ressourcen akzeptiert. Folgendes Beispiel illustriert eine Anfrage mit Accept-Header:

```
GET http://example.org/rest-fs/texte/275
Accept: application/json
```

PUT: PUT erlaubt es, die Repräsentation einer bekannten Ressource durch die Repräsentation im Anfragekörper zu ersetzen. Um Ressourcen zu aktualisieren, wird in vielen Fällen die Operation PUT verwendet. Wird PUT auf eine URI angewandt, an der sich noch keine Ressource befindet, wird die Ressource angelegt. Mit PUT können also Ressourcen an einer vom Client spezifizierten URI angelegt oder geändert werden. Dafür muss die Anfrage aber die spezifizierte URI und die Ressource enthalten. Im Folgenden ein Beispiel zur Veränderung oder Erstellung des Textes 275:

```
PUT http://example.org/rest-fs/texte/275
```

Da PUT eine direkte Auswirkung auf die angefragte Ressource hat, ist es keine sichere Operation. Allerdings ist die Operation idempotent, da nach mehrmaligen Anlegen oder Ändern einer Ressource unter der gleiche URI mit der gleichen Anfrage, das Ergebnis sich

nicht weiter verändert hat. Falls eine Ressource erfolgreich erstellt wurde, sollte ein 201 Statuscode zurückgeliefert werden. Hierbei kann optional der Locationheader mitgesendet werden. Dies ist aber nicht zwingend notwendig, da die Ressouren-ID dem Client bekannt ist, und somit Bandbreite ersparen kann.

POST: Bei POST werden der Ressource Daten übergeben, welche diese dann verarbeitet. In den meisten Fällen wird POST dazu benutzt, um neue Ressourcen - meistens sogar nur Unterressourcen - anzulegen. Bei einer POST-Anfrage wird die URI der übergeordneten Ressource angegeben, welche erweitert werden soll. Im Anfragekörper befindet sich dann der Inhalt der neuen Ressource. Mit POST wird dem Server mitgeteilt, dass er eine neue Ressource anlegen soll, alles Weitere wird vom Server erledigt und der Client hat keinen weiteren Einfluss darauf. Hier ein Beispiel zur Erstellung eines neuen Textes:

```
POST http://example.org/rest-fs/texte/
```

Eine POST-Operation ist nicht sicher, da der Zustand des Servers verändert wird. Auch ist POST nicht idempotent, da der mehrmalige Aufruf der gleichen Anfrage zu mehreren Ressourcen mit dem gleichen Inhalt führt. In unserem Beispiel hätten wir mehrere Texte mit unterschiedlichen IDs, allerdings immer dem gleichen Inhalt. Nach einer erfolgreichen Erstellung einer Ressource sollte ein 201 HTTP-Status mit dem Locationheader zurückgeliefert werden, damit dem Client den Ort der neuen Ressource bekannt ist.

DELETE: Die DELETE-Operation ist dazu da um Ressourcen zu löschen. Die Anfrage enthält dabei die URI der Ressource, welche gelöscht werden soll. Folgendes Beispiel würde den Text 275 löschen:

```
DELETE http://example.org/rest-fs/texte/275
```

DELETE ist nicht sicher, da der Status des Servers durch das Löschen einer Ressource verändert wird. Allerdings sollte DELETE idempotent sein, da das mehrfache Löschen einer Ressource zum gleichen Resultat führt: Die Ressource wurde gelöscht. Allerdings antworten einige Implementierungen von DELETE auf eine Anfrage auf eine nicht existente Ressource mit einer 404 Antwort, was die Operation nicht mehr idempotent macht, da die Antwort verglichen mit der ersten Löschung eine andere ist. Als HTTP-Status-Antworten sollte mit einem 200 geantwortet werden, wenn die Ressource gelöscht wurde. Falls die Löschung nicht sofort vollzogen werden konnte, sollte dem Client ein 202 mitgeteilt werden. Leider kann der Client trotz eines erfolgreichen Löschens nicht sicher sein, dass die Ressource wirklich gelöscht wurde. Einige Implementierungen von DELETE markieren die Ressource nur als gelöscht und stellen sie nicht mehr zur Verfügung, lassen sie aber weiterhin auf dem Server bestehen.

HEAD: HEAD ist eine sehr ähnliche Operation wie GET. Der große Unterschied allerdings ist, dass HEAD keine Repräsentationen zurückliefert, sondern nur die Metadaten der erfragten Ressource. Head bietet somit eine sehr simple Methode sich über Metadaten zu informieren, ohne einen großen Datentransfer anzustoßen. Deshalb wird HEAD oft genutzt, um die Existenz einer Ressource zu überprüfen.

OPTIONS: Die OPTIONS-Operation lässt den Client die Möglichkeiten und Voraussetzungen einer möglichen URI-Anfrage einsehen, ohne diese Anfrage überhaupt zu stellen. Die OPTIONS-Operation ist idempotent und wird auch als safe eingestuft, da eine solche Operation keine Spuren auf dem Server hinterlässt, also seinen Zustand nicht verändert. Wenn ein Server implementiert wird, sollte OPTIONS unterstützt werden, um die Möglichkeiten auf Ressourcen darzustellen und den eigenen Traffic zu senken. Jedoch kann man im Allgemeinen nicht davon ausgehen, dass diese Operation überall unterstützt wird, da OPTIONS sehr selten implementiert und angeboten wird. Eine OPTIONS-Antwort könnte so aussehen:

```
200 OK
Allow: HEAD, GET, PUT, DELETE, OPTIONS
```

TRACE: TRACE bietet die Möglichkeit zu sehen, was am anderen Ende einer Anfrage ankommt und benutzt diese Information dann, um Diagnose-Informationen zu sammeln. Falls die Anfrage gültig ist, sollte die Antwort die gesamte Anfrage Nachricht enthalten. Antworten dieser Anfragen dürfen nicht gecachet werden. Jedoch wird TRACE in der Praxis so gut wie nie verwendet und wird deshalb nur sehr selten unterstützt.

CONNECT: Diese Operation wird in Verbindung mit einem Proxy benutzt, um einen Tunnel aufzubauen. Meist handelt es sich hierbei um einen SSL-Tunnel.

Diese Tabelle zeigt einen Überblick über die HTTP-Operationen und einige ihrer Eigenschaften.

Methode	sicher	idempotent	identifizierbare Ressource
GET	X	X	X
HEAD	X	X	X
PUT		X	X
POST			
OPTIONS	X	X	
DELETE		X	X
CONNECT		X	

Tabelle 2: Eigenschaften der HTTP-Verben

3.2 Welche Prinzipien muss ein REST-Dienst befolgen?

Die meisten Erklärungen, was REST ist, bedienen sich der Forderung, dass die folgenden Prinzipien von dem jeweiligen Dienst erfüllt werden müssen. Die konkrete Implementierung eines Prinzips ist dabei nicht vorgeschrieben, sondern dem Dienst überlassen.

Roy Fielding hat in seiner Dissertation ursprünglich sechs REST-Prinzipien definiert: Client-Server, Stateless, Cache, Uniform Interface, Layered System und Code-On-Demand. Diese wurden mit Zeit von verschiedenen Autoren diskutiert und leicht verändert. Die Ansätze werden nicht von allen Quellen deckungsgleich angeführt. Einigkeit hingegen besteht bei diesen:

3.2.1 Adressierbarkeit/einheitliches Interface(1)

Der Dienst, genauer gesagt die Ressource, muss eindeutig adressierbar sein. Was eine Ressource genau umfasst, wird im nächsten Kapitel ausführlich erläutert. Diese Adressierbarkeit ermöglicht einen konsistenten und einfachen Zugriff auf ein Angebot.

3.2.2 Unterschiedliche Repräsentationen

Ein Zugriff auf einen Dienst liefert nicht ausschließlich das inhaltliche, sich hinter Ressource verbergende, Ergebnis zurück. Ressourcen werden in einer bestimmten Darstellungsform, der sogenannten Repräsentation, zurückgeliefert. Bei einer Repräsentation kann es sich etwa um die Sprache eines Dokuments oder die Kodierungsform (z.B. Zeichensätze wie UTF-8, oder Strukturen wie JSON betreffend) handeln.

3.2.3 Operationen/einheitliches Interface(2)

Zugriffe auf Dienste können verschiedene Aktionen darstellen. So kann etwa ein Dokument angefordert werden - gewissermaßen ein lesender Zugriff. Ebenfalls können aber auch "schreibende" Aktionen ausgeführt werden. Hierbei wird eine Ressource vom Client manipuliert. In der konkreten Implementierung von HTTP werden diese Operationen durch die HTTP-Verben wie GET, POST, PUT und DELETE repräsentiert. Es wird also eine Schnittstelle definiert, die von allen Kommunikationspartnern benutzt und eingehalten werden muss.

3.2.4 Zustandslosigkeit

Die Kommunikation mit einer Ressource geschieht zustandslos. Dies bedeutet, dass bei der verarbeitenden Ressource (also dem Server) keine Informationen über vorangegangene Aktionen implizit vorhanden sind. So ist ein Client etwa nicht davon abhängig, dass der Server den Zustand der Kommunikation intern korrekt gespeichert hat. Vielmehr werden bei jeder Anfrage alle zur Ausführung einer Operation benötigten Informationen mitgeschickt. Dies bedeutet nicht, dass die eigentliche Anwendung keinen Zustand halten kann. Nur die Kommunikation selbst ist hiervon betroffen.

3.2.5 Hypermedia/Linking

Dieses Prinzip besagt, dass neben den Daten selbst auch Metadaten mitgesendet werden (Hypermedia). Diese Metadaten steuern den Zustand des Clients (HATEOAS). Das wohl bekannteste Beispiel für solche Metadaten sind Links auf Webseiten: Diese verknüpfen unterschiedliche Ressourcen, welche unter Umständen auch auf unterschiedlichen Servern liegen, miteinander. Die Verbindung geschieht über Zusatzinformationen in den jeweiligen Repräsentationen einer Ressource.

Folgende Prinzipien werden in der Literatur nicht von allen Autoren erwähnt [4]:

3.2.6 Cacheable

Diese Eigenschaft beschreibt die implizite oder explizite Deklaration eines Web-Dokuments (also etwa der Antwort eines Servers) als zwischenspeicherbar. Durch eine solche eindeutige Kennzeichnung wird verhindert, dass ungültige Antworten weiterbenutzt werden (zum Beispiel weil eine Antwort gespeichert wurde, obwohl das nicht zulässig war). Ferner können durch zwischengespeicherte Elemente die Skalierbarkeit und Performanz gesteigert werden.

3.2.7 Layered System

Bei Schichtensystemen geschieht die Kommunikation transparent für die jeweiligen Endpunkte. Dies bedeutet, dass etwa ein Client nicht bestimmen kann, ob er eine direkte Verbindung zu dem Server hat, oder ob die Kommunikation über Zwischenknoten geleitet wird. Dadurch wird der Einsatz von Zwischenservern ermöglicht, welche der Skalierbarkeit zuträglich sind, etwa indem sie eine angemessene Lastenverteilung bereitstellen. Ferner können Sicherheitsrichtlinien zum Beispiel durch eine Firewall leichter durchgesetzt werden. Dieses Prinzip geht mit der Zustandslosigkeit einher.

3.2.8 Code on Demand

Mit dem optionalen Prinzip des "Code on demand" bezeichnet man die Erweiterung der Clients um Funktionalitäten, die durch die Übermittlung von ausführbarem Code erreicht wurde. Ein synonymer Begriff ist das "client-side scripting".

3.3 HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) [1] ist ein Entwurfsprinzip für REST Architekturen, in dem der Client durch URIs (oder ähnliche Funktionen wie etwa Formulare) navigiert, welche vom Server bereitgestellt werden. In den meisten Fällen werden die URIs über Hypermedia [5] bereitgestellt. Der Client braucht also kein detailiertes Wissen darüber, wie er mit der Anwendung kommunizieren muss. Ein gundlegendes Verständnis über Hypermedia reicht ihm aus um mit dem Server zu kommunizieren. HATEOAS biete auch eine losere Kupplung zwischen Server und Client an. So kann die Schnittstelle des Server geändert werden, ohne dass Änderungen beim Client nötig sind. Der Client benötigt nur eine feste URI als Eingangspunkt und kann dann seine Anfrage durch die bereitgestellten Links zu der gewünschten Ressource durch"klicken". Dies entspricht in etwa der Art wie Menschen über den Webbrowser Webpräsenzen besuchen. Man kommt über eine feste URL auf die Hauptseite und klickt sich dann über Links zur gewünschten Ressource durch.

3.4 Richardson Maturity Model

Das (Richardson) REST Maturity Model [29] ist ein von Leonard Richardson vorgeschlagener Maßstab um zu spezifizieren, wie stark ein Dienst die Prinzipien von REST erfüllt. Dafür definiert er vier Stufen. Für jede dieser Stufen müssen verschiedene Prinzipien von REST erfüllt werden um sie zu erreichen. Die Stufen sind wie folgt gegliedert:

Level 0: RESTles REST Dienste der Stufe 0 verwenden für ihren ganzen Dienst eine einzige URI, die den Endpunkt des Dienstes darstellt. Es wird auch nur ein einziges HTTP-Verb (meistens POST) für sämtliche Aktionen verwendet, was im Falle von POST dafür verwendet wird um SOAP-basierten Payload an den Dienst zu senden. Auch XML-RPC oder Plain Old XML verwenden ähnliche Methoden.

Level 1: Ressourcen Auf dem nächsten Level werden nun Ressourcen verwendet, statt nur eines einzigen Endpunktes für den Dienst. Dies erlaubt die gewünschte Ressource durch den Dienst direkt anzusprechen, da für jede Ressource mindestens eine URI verfügbar ist. Allerdings wird immer noch nur ein einziges HTTP-Verb für den Dienst benutzt.

Level 2: HTTP-Verben Level 2 Dienste stellen nun neben unterschiedlichen Ressourcen und URIs auch die HTTP-Verben zur Verfügung. Es können nun die weiteren HTTP-Verben mit ihren gedachten Eigenschaften und Funktionen verwendet werden, was den Dienst davon wegführt, HTTP nur als Tunnel-Mechanismus zu benutzen. Auch die HTTP-Fehlercodes werden auf diesem Level verwendet.

Level 3: Hypermedia Das letzte Level für REST Dienste, stellt die Verwendung von Hypermedia as the Engine of Application State (HATEOAS) dar. Hierbei enthalten Ressourcen URIs für andere Ressourcen, die für die Anfrage des Clients interessant sein können und die dieser auch weiterverarbeiten kann. Der Client kann nun über die URI-Links von Ressource zu Ressource geführt werden.

4 Vergleich der Designentscheidungen

In diesem Kapitel werden die verschiedenen Designentscheidungen und Best Practices für REST APIs aufgeführt und erläutert. Für jede Designentscheidung oder Best Practice wird beschrieben, wofür sie zu verwenden ist und welche Vorteile sich durch ihre Anwendung ergeben. Wenn es für eine Designentscheidung mehrere technische Möglichkeiten gibt, werden diese Möglichkeiten erläutert und ihre Vor-und Nachteile aufgeführt. Es wird auch dargelegt, welche der Autoren, die sich mit Designentscheidungen für REST APIs beschäftigen, welche Designentscheidung in ihren Werken empfehlen. Für jede Best Practice oder Designentscheidung wird neben der Meinung der Autoren, noch eine Bewertung unsererseits hinzugefügt.

Im Allgemeinen ist die eigentliche Benennung von Identifikatoren, etwa von URIs oder Headerfeldern, irrelevant, da die Kommunikation über Maschinen abgewickelt wird. Aus softwaretechnischen Aspekten, insbesondere der Wartbarkeit und Erweiterbarkeit, bietet es sich jedoch an, für Menschen verständlichere Ausdrücke zu verwenden: Diese erlauben eine leichtere und schnellere Einarbeitung und verringern die Wahrscheinlichkeit, (syntaktische) Fehler in die Anwendung einzubauen.

4.1 Wie versioniere ich eine REST API?

Wie bei jedem Software-Produkt sind auch Schnittstellen späteren Änderungen unterworfen. Diese Änderungen wiegen jedoch schwerer als bei einzelnen Modulen eines Projekts, da sie meist die Verbindung mehrerer einzelner Teile betrifft. Durch die Änderung der (insbesonderen) serverseitigen Schnittstelle werden Clients mit älteren Versionen unter Umständen komplett ausgeschlossen, da Anfragen nicht mehr in der erwarteten Weise beantwortet werden: Die Antwort bleibt gänzlich aus oder liegt in einem Format vor, das der Client nicht verarbeiten kann.

Folgendes Beispiel soll diese Problematik illustrieren: Eine Schnittstelle in der ursprünglichen Definition bietet Zugang zu Einträgen einer Adressdatenbank. Eine Anfrage liefert ein Array zurück, welches den Namen sowie die Telefonnummer enthält:

```
("Hans Müller", "0711 133742")
```

In einer überarbeiteten Variante dieser API soll nun der Name getrennt in Vor- und Nachname zurückgegeben werden, es wird also ein Feld hinzugefügt.

```
("Hans", "Müller", "0711 133742")
```

Ein Client, der seinerseits noch davon ausgeht, dass das erste Feld den kompletten Vor- und Nachnamen enthält, könnte mit der überarbeiteten Variante nicht mehr korrekt umgehen, da die Feldindizes nicht mehr den erwarteten entsprechen. Unter Umständen folgt ein Funktionsverlust, da die Telefonnummer nicht mehr erkannt werden kann.

Weitere mögliche Änderungen an einer API können zum Beispiel sein:

- Änderung eines Feldnamens, etwa bei JSON-Antworten: Umbenennung von "name" in "vorname"
- Änderung eines Feldtyps, etwa die Änderung eines String-Feldes in ein Integer-Feld
- Entfernung eines Feldes
- Änderung des Ressourcenkonzepts: Eine Ressource stellt nicht mehr die ursprüngliche Intention dar. So kann "submitted" ursprünglich das Abgabedatum für einzelne Prüfungen bedeuten. Eine Änderung wäre es, die Bedeutung auf "das Ende einer Prüfung" festzulegen.

All diesen Modifikationen ist gemein, dass sie zu Verarbeitungsproblemen führen können. Die übliche Lösungsstrategie zu diesen Problematiken besteht in der Versionierung der Schnittstelle. Dadurch können Kommunikationspartner, die nur eine ältere Variante unterstützen, weiterhin bedient werden, während gleichzeitig die neue Version der Schnittstelle parallel angeboten wird.

Bezieht man diese möglichen Probleme frühzeitig in die Planung seiner Schnittstelle mit ein, erleichtert dies spätere Änderungen und deren Verwaltung. Falls eine neue Version einer Schnittstelle angeboten werden soll, muss der Server einen Mechanismus bereitstellen, um die Änderung bzw.

vorhandene Versionen den Clients mitzuteilen. Der Client kann daraufhin seine Anfrage anpassen. Unter Umständen ist dies jedoch erst nach einem Update möglich.

Im Folgenden werden zwei grundsätzlich verschiedene Lösungsansätze vorgestellt, um eine Versionierung zu realisieren. Nachfolgende Tabelle bietet einen Überblick darüber, welche Autoren in ihren Veröffentlichungen eine Versionierung über den Header oder über die URL empfehlen.

Header	URL
Fredrich [42]	Jaucker [36]
Dhall [21]	Hunter [40]
Mulloy [14]	Sahni [43]
Klabnik [38]	Seely [35]
Wiliams [32]	Allemaraju [11]
Freed & Klensin [24]	
Tilkov [41]	

Tabelle 3: Versionierung in der Literatur

4.1.1 Lösungsansatz 1: Versionierung mittels Header

Ein Ansatz zur Versionierung von REST APIs basiert nun darauf, unterschiedliche Versionen einer Ressource als unterschiedliche Repräsentationen zu modellieren. Clients können dann mit den üblichen Mechanismen der Content Negotiation entscheiden, mit welcher Version bzw. Repräsentation sie interagieren möchten. Nachfolgend werden zwei Möglichkeiten, dies in HTTP umzusetzen, vorgestellt.

Die Benutzung eines eigens definierten Headers (custom header) Hierbei definiert man sich für seine API ein eigenes Feld, etwa "fs-version":

```
Anfrage:
GET http://example.org/rest-fs/texte/275
fs-version: 2

Antwort:
HTTP/1.1 200 OK

{"id":"275","name":"Einleitung","position":"2"}
```

Verwendung des Vendor-Feldes Ganz ähnlich wie oben verhält es sich mit den Vendor-Feld. Dies geschieht über die korrekte Benutzung von "Accept" im Anfrage-Header. Der Austausch sieht dabei wie folgt aus:

```
Anfrage:
GET http://example.org/rest-fs/texte/275
Accept: application/vnd.fs-ausarbeitung.v2+json

Antwort:
HTTP/1.1 200 OK
Content-Type: application/json; version=2

{"id":"275","name":"Einleitung","position":"2"}
```

Ebenso ist es möglich, den MIME-Typ zu parametrisieren. Leider wird diese Variante noch nicht von allen Web-Frameworks unterstützt. Möchte man etwa die Version 2 der Ressource als JSON erhalten, sieht der Ablauf folgendermaßen aus:

```
Anfrage:
GET http://example.org/rest-fs/texte/275
Accept: application/json; version=2

Antwort:
HTTP/1.1 200 OK

Content-Type: application/json; version=2
{"id":"275","name":"Einleitung","position":"2"}
```

Durch die Verwendung des Headers können Nebeneffekte auftreten. Zum Beispiel wird die Erkennbarkeit der Version für den Menschen erschwert, da sie nicht sofort ersichtlich ist und der Nutzer erst in den Header hineinschauen muss. Bei der Versionierung über URLs wäre das nicht der Fall, da die Versionsbezeichnung bereits in der URL selbst enthalten ist, wodurch sie dem Nutzer auf einen Blick ersichtlich ist.

4.1.2 Lösungsansatz 2: Versionierung mittels URL

Die zweite Möglichkeit, eine Versionierung zu Realisieren, besteht im Design der URLs. Hierbei wird eine URL nicht nur als "syntaktischer Name" gesehen, sondern als Entität mit interner Struktur. So werden bei diesem Ansatz die beiden URLs

```
http://example.org/rest-fs/texte/275/v1
http://example.org/rest-fs/texte/275/v2
```

nicht als grundsätzlich verschieden gesehen, da sie einem bestimmten Aufbau folgen: *<Protokoll><eigentliche Ressource>/<Version>*.

Weitere Möglichkeiten, eine Versionierung über die URL zu bewerkstelligen, besteht in der Verwendung von:

Parametern: Hierbei wird an die eigentliche Ressource ein oder mehrere Parameter angehängt. Dieser Parameter wird bei URLs auch "query" genannt. Ein Beispiel für eine Anfrage mit query:

```
GET http://example.org/rest-fs/texte/275?version=2
```

Subdomains: Diese Variante der URLs bietet sich insbesondere bei größeren Versionssprüngen (Major Releases) an. Hierbei wird die Versionsnummer bzw. Bezeichnung als Subdomain vorangestellt. Im Folgenden wird etwa "v1" verlangt:

```
GET http://v1.example.org/rest-fs/texte/275
```

4.1.3 Vor- und Nachteile

Beide Möglichkeiten zur Versionierung bieten verschiedene Vor- und Nachteile. Im Folgenden werden einige für Versionierung mittels URLs und Header beschrieben.

- **Vorteile der Versionierung mittels URL:**

- Bessere Erkennbarkeit der Version: Dem Nutzer ist sofort ersichtlich, um welche Version es sich handelt; er muss nicht zusätzlich in den Header sehen.

- Der Zwang zur Angabe einer Version: Um auf die entsprechende Ressource zugreifen zu können, muss die gewünschte Version zwangsläufig angegeben werden. Es kann also zu keinen Mehrdeutigkeiten kommen, da der Client keine explizite Version anfordert. Dies kann insbesondere dadurch erreicht werden, indem man die Version mitten in der URL einbaut, etwa so:

```
http://example.org/rest-fs/v1/texte/275
```

- **Nachteile der Versionierung mittels URL:**

- Übersichtlichkeit vieler verknüpfter Versionen: Sollte eine Ressource nicht nur in unterschiedlichen Versionen, sondern auch in unterschiedlichen Repräsentationen bereitgestellt werden, sollten sowieso Header verwendet werden. Geschieht dies nicht, leidet die Übersichtlichkeit stark. Klarer wird dies an einem Beispiel:

  Die Ressource soll in den drei Sprachen Englisch, Deutsch und Französisch und zusätzlich in den Versionen 1, 2 und 3 bereitgestellt werden. Mögliche URLs wären also:

  ```
  http://example.org/rest-fs/en/v1/texte/275
  http://example.org/rest-fs/en/v2/texte/275
  http://example.org/rest-fs/en/v3/texte/275
  http://example.org/rest-fs/de/v1/texte/275
  http://example.org/rest-fs/de/v2/texte/275
  http://example.org/rest-fs/de/v3/texte/275
  http://example.org/rest-fs/fr/v1/texte/275
  http://example.org/rest-fs/fr/v2/texte/275
  http://example.org/rest-fs/fr/v3/texte/275
  ```

  Hier wird schnell klar, dass die eigentliche Ressource und ihre Repräsentationen nicht strikt von einander abgegrenzt sind.

- Unterscheiden sich die Versionen nicht ausschließlich in Ganzzahlen, wird die entsprechende URL schnell unübersichtlich und für Menschen schwer lesbar:

  ```
  http://example.org/rest-fs/v29.3.4-de.2/texte/275
  ```

- **Vorteile der Versionierung mittels Header:**

- Der wesentliche Vorteil bei der Verwendung von Headern liegt in der Entkoppelung der Version von der eigentlichen Ressource. Dies gelingt durch die Auffassung einer Version als unterschiedliche Repräsentation derselben Ressource.

- **Nachteile der Versionierung mittels Header:**

- Durch die Verwendung des Headers können Nebeneffekte auftreten. Zum Beispiel wird die Erkennbarkeit der Version für den Menschen erschwert, da sie nicht sofort ersichtlich ist und der Nutzer erst in den Header hineinschauen muss. Bei der Versionierung über URLs wäre das nicht der Fall, da die Versionsbezeichnung bereits in der URL selbst enthalten ist, wordurch sie dem Nutzer auf einen Blick ersichtlich ist.

4.1.4 Diskussion und Bewertung

In den vorigen Absätzen wurden die verschiedenen Möglichkeiten und Standpunkte der Autoren vorgestellt. Nun möchten wir darlegen, welchen Standpunkt wir vertreten. Die Versionierung einer RESTful API sollte grundsätzlich unabhängig von der URL sein, da keine gänzlich andere Ressource angefordert wird, sondern nur eine andere Version. Ebenfalls ist durch die Verknüpfung einzelner Ressourcen durch Hypermedia-Inhalte (also etwa Links auf Webseiten) die Menschenlesbarkeit bei der Benutzung der jeweiligen Ressource unerheblich. Daher ist in unseren Augen die Versionierung über Header zu empfehlen.

4.1.5 Verwendung in der Praxis

In der Praxis kommen die beiden Lösungsansätze sehr ungleich verteilt vor. Es wird wesentlich häufiger die Versionierung über URLs verwendet. Nachfolgend eine Aufzählung ausgewählter APIs und deren Art der Versionierung.

Schnittstellenname	Versionierungsart	Versionierungsbeispiel
Twilio	Datum in der URL	http://.../2010-04-01/Accounts/AccountSid/Calls
Twitter	URL	http://api.twitter.com/1/users/show/noradio.xml
Atlassian	URL	http://host:port/context/rest/upm/1/...
Google Search	URL	https://developers.google.com/.../v1/...
Facebook	URL	http://graph.facebook.com/v1.0/me
Bing Maps	URL	http://dev.virtualearth.net/REST/v1/Locations/CA/...
Netflix	Parameter in URL	http://api.netflix.com/catalog/titles/series/70023522?v=1.5
Google data	Parameter in URL	http://...?v=X.0
Digg	URL	http://services.digg.com/2.0/comment.bury
Delicious	URL	https://api.del.icio.us/v1/posts/update
Last FM	URL	http://ws.audioscrobbler.com/2.0/
LinkedIn	URL	http://api.linkedin.com/v1/people/ /connections
Foursquare	URL	https://api.foursquare.com/v2/...
Freebase	URL	https://www.googleapis.com/freebase/v1/...
Paypal	Parameter in URL	http://...&VERSION=XX.0
Twitpic	URL	http://api.twitpic.com/2/upload.format
Etsy	URL	http://openapi.etsy.com/v2
Tropo	URL	https://api.tropo.com/1.0/sessions
Tumblr	URL	api.tumblr.com/v2/user/
Openstreetmap	URL	http://server/api/0.6/changeset/create
Ebay	Parameter in URL	http://open.api.ebay.com/shopping?version=713
Groupon	URL	http://api.groupon.com/v2/channels/...
Bitly	URL	https://api-ssl.bitly.com/v3/shorten
Disqus	URL	https://disqus.com/api/3.0/posts/remove.json
Drop Box	URL	https://api.dropbox.com/1/oauth/...
Youtube data	URL	https://www.googleapis.com/youtube/v3
Github API	Accept Header	Accept: application/vnd.github.v3+json
Microsoft Azure	Custom Header	x-ms-version: 2011-08-18

Tabelle 4: Versionierung in ausgewählten Beispielen

4.1.6 Weitere verwandte Frage-/Problemstellungen

Aus der Möglichkeit der Versionierung ergeben sich wiederum weitere Fragestellungen und mögliche Probleme, denen sich REST-API Entwickler stellen müssen. Der Entwickler einer REST-API muss sich überlegen, nach welchen Änderungen eine neue Version erstellt werden muss und welche Änderungen keine neue Version mit sich ziehen. Auch sollte der Entwickler überdenken, welche Version er auf Anfragen zurückgeben möchte, welche keine Versionierung benutzen und wie er reagiert, wenn sich notwendige Änderungen in seiner URL-Struktur ergeben.

4.2 Wie benenne ich eine Ressource?

Bei der Art und Weise, wie Ressourcen benannt werden, herrscht weitgehend Einigkeit. Grundsätzlich gilt bei Ressourcennamen, dass die Semantik hinter der Bezeichnung für Maschinen unerheblich ist. Unter softwaretechnischen Aspekten wie der Wartbarkeit und Erweiterbarkeit jedoch wird dringend dazu geraten, folgende Ratschläge umzusetzen.
Nachfolgende Tabelle bietet einen Überblick darüber, welche Autoren in ihren Veröffentlichungen auf die Ressourcenbenennung eingehen.

Allamaraju [11]
Massé [30]
Roth [27]
Richardson & Ruby [33]
Tilkov [41]
Fredrich [42]
Dhall [20]
Mulloy [13]
Jauker [36]
Hunter [40]
Sahni [43]

Tabelle 5: Ressourcenbenennung in der Literatur

- Selbstbezeichnende Ressourcennamen: Die Bezeichnung einer Ressource sollte möglichst immer selbstbezeichned sein. Ein Negativbeispiel hierzu ist die willkürliche Verwendung der zur Verfügung stehenden Zeichen, etwa:

  ```
  GET http://example.com/j2CAd23A05dF
  ```

  Besser wäre eine Bezeichnung, aus welcher hervorgeht, was sich hinter der Ressource verbirgt. Also zum Beispiel:

  ```
  GET http://example.com/todo-liste
  ```

- Substantive verwenden: Eine Ressource sollte durch ein Substantiv bezeichnet werden. Dies ermöglicht eine konsistente Verwendung der bekannten HTTP-Verben, da sich diese immer auf das Substantiv bzw. die Ressource beziehen. Würden statt der Substantive Verben benutzt, so stellt man bei der letztlichen Verwendung schnell fest, dass keine Objekte (Ressourcen) mehr modifiziert werden können, sondern nur eine bestimmte Aktion auf ihnen ausgeführt werden kann. Ein Beispiel liefert ein Forum. Soll auf ein Thema mit der ID 1234 zugegriffen werden, so könnte das folgendermaßen geschehen:

  ```
  GET http://example.com/getthread/1234
  ```

  Die Probleme werden deutlich sichtbar, sobald das HTTP-Verb verändert wird. Möchte man genau dieses Thema bearbeiten und dabei Daten an den Server schicken, sähe die Operation wie folgt aus:

  ```
  POST http://example.com/getthread/1234
  ```

  Hier wird der Bruch zwischen dem Verb "POST", welches für eine Veränderung eingesetzt wird, und dem "get" in der Ressourcenbezeichnung selbst klar. Besser wäre hier - entsprechend des Grundgedankens einer Ressource - der Zugriff auf den Thread selbst. Die Zugriffsart (etwa lesender Zugriff oder Modifikation) wird dann über das HTTP-Verb geregelt:

  ```
  GET http://example.com/threads/1234
  POST http://example.com/threads/1234
  ```

- Plural verwenden: Oft wird durch eine Ressource nicht nur ein einzelnes Element abgebildet, sondern eine Menge von Elementen. Daher bietet sich die Verwendung des Plurals an. Statt

  ```
  GET http://example.com/user/1529
  ```

  wird also besser

  ```
  GET http://example.com/users/1529
  ```

  verwendet.
  Von dieser Best Practice kann natürlich abgesehen werden, wenn garantiert ist, dass nur ein Element verfügbar sein wird. Ein Beispiel könnte eine abgelegte Serverkonfiguration sein:

  ```
  GET http://example.com/configuration
  ```

4.2.1 Diskussion und Bewertung

Obwohl es für die API egal ist, wie die Ressourcen aufgebaut sind, sind wir als Softwaretechniker der Meinung, dass die Autoren damit recht haben, eine für Menschen lesbare Struktur in die Ressourcen zu bringen, da diese Struktur den Wartungs- und Weiterentwicklungsaufwand stark verringern kann.

4.3 Wie biete ich verschiedene Repräsentationen einer Ressource an?

Der durch eine Ressource angebotene Inhalt kann in verschiedenen Formaten angeboten werden. Dies ermöglicht, dass der Client die Wahl des Formates selbst treffen kann, was ihm die Möglichkeit bietet, nur solche Formate zu erhalten, welche er auch sicher verarbeiten kann. Daraus ergibt sich für den Entwickler einer Schnittstelle die Frage, wie er eine solche Auswahl bereitstellen kann. Die verschiedenen Verfahren werden im Folgenden beschrieben.

Diese Tabelle bietet einen Überblick darüber, welche Autoren in ihren Veröffentlichungen verschiedene Formate der Repräsentationen über den Header oder über die URL bevorzugen.

Header	URL
Allamaraju [11]	Allamaraju [11]
Massé [30]	Richardson & Ruby [33]
Freed & Klensin [24]	Fredrich [42]
Richardson & Ruby [33]	Sahni [43]
Jauker [36]	
Dhall [21]	
Erl et al. [39]	
Webber et al. [28]	

Tabelle 6: Repräsentationsformate in der Literatur

Die nun behandelten Beispiele gehen von den zwei gebräuchlichen Formaten XML und JSON aus.
Grundsätzlich bieten sich zwei unterschiedliche Realisierungen für die Unterstützung mehrerer Formate an: Die Verwendung eines entsprechenden (Anfrage-)Headers und die Nutzung der URL.

4.3.1 Lösungsansatz 1: Formatangabe im Header

Bei dieser Variante wird der in HTTP spezifizierte Accept-Header, welcher zur Content Negotiation benutzt wird, verwendet. Dadurch wird ein Format als unterschiedliche Repräsentation derselben Ressource gesehen. Eine einfache Anfrage an eine Ressource, welche eine Mitarbeiterliste modelliert, könnte zum Beispiel wie folgt aussehen. Diese Liste wird hierbei im JSON-Format verlangt.

```
Anfrage:
GET http://example.com/mitarbeiterliste
Accept: application/json

Antwort:
HTTP/1.1 200 OK
Content-Type: application/json

{{"id":"1","vorname":"Peter","Meier","abteilung":"SGA5"}
{"id":"2","vorname":"Maria","Engelhard","abteilung":"PA2"}}
```

Über den Header ist es zudem möglich, eine default-Repräsentation (in diesem Fall also ein default-Format) zurückzugeben, sofern kein Accept-Header in der Anfrage definiert wurde. Dadurch erhält Client entweder eine Repräsentation, die er verarbeiten kann, oder einen beschreibenden Fehler zurück.

4.3.2 Lösungsansatz 1: Formatangabe in der URL

Die andere Möglichkeit, unterschiedliche Formate anzubieten, besteht in der Verwendung der URL. Hierbei wird nicht eine unterschiedliche Repräsentation verlangt, sondern (im REST-Sinne) eine komplett andere Ressource. Dies kann auf zwei Arten geschehen:

- Durch die Angabe einer formatspezifischen Endung (zum Beispiel .json):

```
Anfrage:
GET http://example.com/mitarbeiterliste.json

Antwort:
HTTP/1.1 200 OK
Content-Type: application/json

{{"id":"1","vorname":"Peter","Meier","abteilung":"SGA5"}
{"id":"2","vorname":"Maria","Engelhard","abteilung":"PA2"}}
```

- Durch die Angabe eines formatspezifischen Parameters (zum Beispiel: "type=json"):

```
Anfrage:
GET http://example.com/mitarbeiterliste?type=json

Antwort:
HTTP/1.1 200 OK
Content-Type: application/json

{{"id":"1","vorname":"Peter","Meier","abteilung":"SGA5"}
{"id":"2","vorname":"Maria","Engelhard","abteilung":"PA2"}}
```

Auch bei der Verwendung von URLs ist es möglich, ein default-Format anzubieten. Darauf wird zugegriffen, indem die jeweilige Endung weggelassen wird. Beim default-Format "JSON" etwa liefern folgende zwei Anfragen denselben Inhalt zurück:

```
GET http://example.com/mitarbeiterliste
GET http://example.com/mitarbeiterliste.json
```

4.3.3 Vor- und Nachteile

Die beiden Varianten zur Angabe verschiedener Formate haben eigene Vor- und Nachteile, welche im Folgenden beschrieben werden.

- **Vorteile der Formatangabe im Header:**

- Der wesentliche Vorteil bei der Verwendung von Headern liegt in der Entkoppelung des Formats von der eigentlichen Ressource. Dies gelingt durch die Auffassung eines Formats als unterschiedliche Repräsentation dersel Ressource ben Ressource. Es werden also nicht mehrere Ressourcen für denselben Inhalt benötigt.

- **Nachteile der Formatangabe im Header:**

- Als Nachteile wäre zu erwähnen, dass durch den Header die Lesbarkeit für Menschen eingeschränkt wird. Da die Formatangabe nicht sofort ersichtlich ist muss, der Nutzer erst in den Header hineinsehen, um sie zu erkennen.

- Beim Verwenden eigener Formate, muss für die Benutzung des Accept-Headers ein eigener Formattyp (custom type) definiert werden, da kein Standardtyp verfügbar ist. Dies macht die Schnittstelle unnötig komplexer.

- **Vorteile der Formatangabe in der URL:**

- Dem Nutzer ist sofort ersichtlich, um welches Format es sich handelt; er muss nicht zusätzlich in den Header sehen. Es handelt sich hierbei also um ein Format, was insbesondere die Benutzung simpler Client-Programme wie "curl" vereinfacht.

- Um auf die entsprechende Ressource zugreifen zu können, muss das gewünschte Format zwangsläufig angegeben werden. Es kann also zu keinen Mehrdeutigkeiten kommen, da der Client kein explizites Format anfordert.

- **Nachteile der Formatangabe in der URL:**

- Mehrere Ressourcen müssen erstellt werden, um das jeweilige Format abzudecken. Zum Beispiel eine für JSON und eine für XML:

```
http://example.com/mitarbeiterliste.json
http://example.com/mitarbeiterliste.xml
```

- Die Content Negotiation im Hypertext Transfer Protocol legt fest, dass die Wahl der Repräsentation einer Ressource über Accept-Header läuft. Wird dieses Verfahren ausgehebelt, indem man die Ressource mit der Repräsentation mischt, wird streng genommen das Protokoll misachtet.

4.3.4 Diskussion und Bewertung

Unserer Auffassung nach sollten die verschiedenen Repräsentationen einer Ressource nicht in der URL, sondern im Header spezifiziert werden. Dies ist in der strikten Trennung von Ressourcenidentität und Ressourcenrepräsentation begründet: Wird das Format bzw. die Repräsentation in der URL angegeben, ist diese Trennung nicht mehr gegeben - Dem HTTP-Protokoll nach wird dadurch eine ganz andere Ressource angesprochen. Erst die Befolgung eines Protokolls ermöglicht eine fehlerfreie Kommunikation aller Kommunikationspartner.

4.4 Wie soll ich meine URL entwerfen?

Da die REST APIs über HTTP laufen, werden als Identifikatoren der Ressourcen und der API URLs verwendet. Genaugenommen ist es für die Entwicklung einer REST API egal, wie diese URLs aufgebaut sind, da Anfragen und Antworten von Maschinen kommen. Den Client interessiert es technisch gesehen nicht, ob die URL

```
http://example.org/afsfsdfa/5/afasdf/
```

lautet, oder ob

```
http://example.org/cars/17/drivers/
```

die gewünschte Ressource enthält, so lange die URLs einzigartig sind.

Allerdings sind sich die Autoren einig, dass auch für URLs Strukturen bestehen sollten und diese Strukturen als Best Practice bei der Entwicklung von REST APIs eingehalten werden sollten. Sie begründen diese Strukturen dadurch, dass die URLs auch für den Menschen lesbar sein sollen.

Nachfolgende Tabelle bietet einen Überblick draüber, welche Autoren in ihren Veröffentlichungen auf die Struktur von URLs eingehen.

Allamaraju [11]
Massé [41]
Roth [27]
Tilkov [41]
Richardson & Ruby [33]
Dhall [20]
Fredrich [42]
Jauker [36]
Mulloy [13]
Sahni [43]

Tabelle 7: URL-Struktur in der Literatur

- URL der API: Die URL, durch die die API erreicht werden kann, sollte sehr einfach und intuitiv sein. Ein gutes Beispiel wäre

  ```
  http://example.org/api/
  ```

  oder

  ```
  http://api.example.org
  ```

- Zwei Basis URLs pro Ressource: Es sollte für jede Ressource zwei URLs geben. Eine URL um alle Werte der Ressource zu erhalten und eine URL um einen spezifischen Wert der Ressource zu erhalten. Dadurch kann unterschieden werden, ob direkt eine Ressource angefragt wird, oder mehrere Ressourcen das Ziel der Anfrage sind.

  ```
  DELETE http://example.org/fachstudientexte/135
  ```

  würde nur den Text mit der ID 135 löschen, während

  ```
  DELETE http://example.org/fachstudientexte/
  ```

  alle Texte löschen würde. So kann allein durch die Struktur der URL schon vermieden werden, dass ungewollte Aufrufe entstehen.

- Assoziationen: Wenn Ressourcen durch Assoziationen verknüpft werden sollen, sollten ihre URLs besondern intuitiv sein, damit auf einen Blick klar ist, was genau angefragt wird. Mit

  ```
  GET http://example.org/trainers/14/training/114
  ```

  ist sofort klar, dass das Training mit der ID 114 von dem Trainer mit der ID 14 angefragt wird. Durch Assoziationen kann eine Tiefe an Ressourcentraversionen entstehen. Im obigen Beispiel liegt die Tiefe bei zwei, die erste Tiefe durch den Trainer und die zweite durch das Training selbst. Man sollte darauf achten, diese Tiefe an Traversionen nicht zu tief werden zu lassen, da sonst die URL nicht mehr intuitiv und unübersichtlich wird.

4.4.1 Diskussion und Bewertung

Obwohl es für die API egal ist, wie die URL aufgebaut ist, sind wir als Softwaretechniker der Meinung, dass die Autoren damit recht haben, eine für Menschen lesbare Struktur in die URLs zu bringen, da diese Struktur den Wartungs- und Weiterentwicklungsaufwand um einiges senken kann.

4.5 Wie mache ich meine Schnittstelle effizienter?

Um Bandbreite zu sparen und den Zugriff auf die API effizienter zu machen, bieten sich dem Entwickler mehrere Möglichkeiten. In der Literatur werden davon vier aufgegriffen und thematisiert. Diese vier Möglichkeiten werden nun vorgestellt. Es handelt sich dabei um Partielle Antworten, Filterung, HTTP-Caching und Pagination.

Die nachfolgende Tabelle bietet einen Überblick draüber, welche Autoren sich in ihren Veröffentlichungen mit der Verbesserung der Effizienz von REST APIs befassen.

Allamaraju [11]
Jansen [25]
Richardson & Ruby [33]
Fredrich [42]
Mulloy & Swiber [19]
Mulloy [12] Dhall [21]
Jauker [36]
Hunter [40]
Sahni [43]
Webber et al. [28]
Erl et al. [39]

Tabelle 8: API-Effizienz in der Literatur

4.5.1 Partielle Antworten (Partial Response)

Um einerseits Bandbreite zu sparen und andererseits Clienten mit einer beschränkten Benutzeroberfläche (zum Beispiel bei mobilen Endgeräten) zu ermöglichen, die angefragten Daten sinnvoll anzeigen zu können, ist es nicht nötig, alle Teile einer Ressource zu übertragen, sondern nur jene, welche der Client wirklich braucht. Insofern sollte die REST API in der Lage sein, Anfragen für partielle Antworten zuzulassen und diese auch dementsprechend beantworten zu können. Dazu muss die URL der Ressource so entworfen werden, dass in der Anfrage über Parameter die gewünschten Teile der Ressource erfragt werden können. Als häufige Implementierung findet sich bei den Autoren die Nutzung von Feldparametern in der URL (query-Teil der URL), welche durch Kommas getrennt werden.

```
GET example.com/songs?fields=id,name,description
```

Betrachtet man große APIs, wie die von Google oder Facebook, sieht man, dass diese Schnittstellen ihre partielle Antworten auf diese Weise geben.

Bei Facebook sieht eine URL mit Feldparametern so aus:

```
/peter.peterson/friends?fields=id,name,picture
```

während eine URL bei Google folgendes Schema vorhanden ist:

```
?fields=title,media:group(media:thumbnail)
```

Es ist allerdings wichtig zu beachten, dass partielle Antworten nur optional sind und nicht bei jeder Anfrage gefordert werden.

4.5.2 HTTP-Caching

Um bei Anfragen, die sich auf unveränderte Ressourcen beziehen, nicht immer die Repräsentation der Ressource als Antwort zu senden, bietet HTTP von Haus aus zwei Möglichkeiten für Caching an: `ETags` und `Last-Modified`. Auch die HTTP-Statuscodes beinhalten den Fall einer Anfrage auf eine unveränderte Ressource, sodass der Client die Bedeutung der Antwort schon in ihrem Statuscode sieht.

`ETags` sind Identifikatoren, welche vom Web Server einer Ressource und deren URL zugeordnet werden. Sollte sich die Ressource ändern, wird ihr ein neuer `ETag` zugeordnet. Diese Identifikatoren können einfache Revisionsnummern sein, eigene Identifikatoren oder berechnete Werte der Ressource, wie deren Hash oder Checksumme. `HTTP 1.1` bietet die Headerfelder `ETag` und `If-None-Match` an, welche in den Anfragen angegeben werden können. Wenn eine Ressource zum ersten Mal angefragt wird, sendet der Server den `ETag-Wert` der Ressource im `ETag-Headerfeld` mit. Möchte der Client nun testen, ob sich die Ressource geändert hat, benutzt er im `If-None-Match-Headerfeld` seiner Anfrage den `ETag-Wert` der Ressource. Wenn die Werte übereinstimmen, wird der Server den Inhalt der Ressource nicht wieder übermitteln.

Eine Anfrage einer Ressource, deren Antwort mit `ETag`, eine Überprüfung ob sich die Ressource geändert hat und dessen Antwort würde wie folgend aussehen:

```
Anfrage:
GET http://example.org/devil

Antwort
HTTP/1.1 200 OK
ETag: "666"

Anfrage:
GET http://example.org/devil
If-None-Match: "666"

Antwort:
HTTP/1.1 304 Not Modified
```

`Last-Modified` funktioniert genauso wie `ETags`, nur dass hier Zeitstempel verwendet werden. Die Antwort auf eine angefragte Ressource enthält `Last-Modified-Headerfeld`, welches den Zeitstempel der letzten Änderung der Ressource enthält. Eine erneute Anfrage würde den `If-Modified-Since-Header` beinhalten, dessen Wert mit dem Zeitstempel der Ressource abgeglichen wird. Nur wenn diese Werte nicht gleich sind, würde die Ressource in die Antwort gepackt werden.

Eine Anfrage einer Ressource, deren Antwort mit `Last-Modified`, eine Überprüfung ob sich die Ressource geändert hat und dessen Antwort würde wie folgend aussehen:

```
Anfrage:
GET http://example.org/dog

Antwort
HTTP/1.1 200 OK
Last-Modified: Thu, 10 Jan 2013 19:43:31 GMT

Anfrage:
GET http://example.org/dog
If-Modified-Since: Thu, 10 Jan 2013 19:43:31 GMT

Antwort:
HTTP/1.1 304 Not Modified
```

Hierbei ist noch zu beachten, dass HTTP mehrere Zeitformate unterstützt, welche von der API alle erkannt werden sollten.

4.5.3 Filterung

Durch das Filtern von Ergebnissen, kann schon bei der Anfrage bestimmt werden, dass nur Ressourcen, welche ein gewünschtes Attribut besitzen, angezeigt (und damit übertragen) werden. Dadurch wird einerseits die genutzte Bandbreite reduziert und der Client muss bei sich keine weitere Filterung vornehmen.

Die Autoren nehmen, wie bei den partiellen Antworten, das Filtern über Anfrageparameter (query paramter) in der URL vor. Dadurch bleibt die Grund-URL einer Ressource simpel, es können jedoch komplexere Anfragen darauf angewendet werden. Dies sollte beim Entwurf der API beachtet werden.

```
GET http://example.org/dog?color=red
```

würde dem Client nur jene Hunderessourcen übertragen, welche das Attribut `red` besitzen.

4.5.4 Paginierung (Pagination)

Wenn eine Anfrage auf eine große Menge von Ressourcen und Daten gemacht wurde, können die Ergebnisse die Benutzeroberfläche leicht mit Informationen überfluten und so unübersichtlich machen. Durch die Paginierung werden die Ergebnisse auf eine überschaubare Menge an Einträgen pro Seite limitiert. Für die Art der Paginierung stellen die verschiedenen Autoren zwei Möglichkeiten vor: die Benutzung von `Anfrageparametern in der URL` und die Nutzung des `HTTP-Link-Headers`. Allerdings geben sie jeweils nur eine Art davon an und auch dies ohne Argumente, warum diese implementiert werden sollte.

- **Anfrageparameter in der URL**
  Hierbei wird die Anzahl der Elemente, die angezeigt werden sollen, über Parameter in der URL angeben. Dabei gibt es wiederum verschiedene Namenskonventionen, welche von verschiedenen APIs benutzt werden. Möchte man zu Beispiel die Ergebnisse 50 bis 75 einer Menge an Teekannen bekommen würde dies wie folgt aussehen:

  ```
  Facebook: /teapots?offset=50&limit=25
  Twitter: /teapots?page=3&rpp=25
  LinkedIn: /teapots?start=50&count=25
  ```

- **HTTP-Link-Header**
  Durch die Nutzung des Link-Headers kann die API dem Client eine begrenzte Menge an Ergebnisse zurückgeben. Hierbei werden jedoch die weiterführenden Links direkt erstellt und dies muss nicht von Hand gemacht werden, was die Fehleranfälligkeit und den Arbeitsaufwand verringert. Folgendes Beispiel aus der `GitHub-API` [9] stellt einen korrekt genutzten `HTTP-Link-Header` dar:

  ```
  Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
  ```

Obwohl sich die Autoren darin einig sind, dass es einen Standardwert für die Paginierung geben soll (wenn keine in der Anfrage angegeben ist), ist die Anzahl der angezeigten Ergebnisse bei jedem Autor unterschiedlich.

4.5.5 Diskussion und Bewertung

Obwohl alle hier genannten Praktiken nicht notwendig sind um eine API RESTful zu machen, da sie keines der Grundprinzipien von REST erfüllen (allerdings auch keinem widersprechen), sind wir mit den Autoren einer Meinung, dass sie wichtig sind und in den Entwurf eine REST API gehören, da sie Vorteile für die API bringen, ohne den Grundprinzipien von RESTful APIs zu widersprechen.

4.6 Soll ich HATEOAS benutzen?

Obwohl die Hyperlinks in der Definition von REST als elementare Eigenschaft aufgeführt werden, sind sich die Autoren über deren Nutzung uneinig. In dieser Tabelle werden jene Autoren aufgeführt die sich für oder gegen die Nutzung von HATEOAS aussprechen.

Für HATEOAS	Gegen HATEOAS
Mulloy [16]	Sahni [43]
Mulloy [17]	
Jaucker [36]	
Fielding [34]	
Fredich [42]	
Klabnik [37]	
Tilkov [41]	
Richardson & Ruby [33]	

Tabelle 9: Nutzung von HATEOAS in der Literatur

Roy Fielding bezeichnet in seinem Blog [34] Hyperlinks als einen der essentiellen Teile einer REST API und spricht sich stark für deren Nutzung aus. Andere Autoren sind der Meinung, dass HATEOAS ebenfalls eine wichtige Rolle in REST APIs spielen, und durch geringen Aufwand zu einer deutlichen Verbesserung der API führen. Jedoch weisen sie auf die Problematik hin, welche bei Sahni [43] als Grund der Ablehnung von HATEOAS geführt wird. Er spricht sich gegen die Nutzung von HATEOAS aus, da beim Surfen durch Links, der Mensch die Entscheidungen während der Laufzeit trifft, wohin gegen viele APIs diese Entscheidungen während der Entwicklungszeit treffen müssen. Die APIs könnten diese Entscheidungen zwar auch während der Laufzeit treffen, allerdings seien die Implementierungen momentan noch nicht soweit, um dies auch fehlerfrei auszuführen, wenn Änderungen an der API vorgenommen wurden.

4.6.1 Diskussion und Bewertung

Wir sind der Meinung, dass HATEOAS in RESTful APIs benutzt werden sollten, da Hypermedia eines der Grundprinzipien von REST ist und APIs dadurch eine wenig arbeitsintensive Verbesserung erhalten. Das Risiko der eventuellen Nichtfunktionalität, welches bei signifikanten Änderungen der API auftreten kann, würden wir im Vergleich zu den Vorteilen in Kauf nehmen.

4.7 Wie soll meine Schnittstelle dokumentiert werden?

Zum Design und Entwurf einer API, sei sie nun RESTful oder auch nicht, gehört immer eine vollständige, verständliche und gute Dokumentation. Darin sind sich alle Autoren (und auch jeder andere Softwareentwickler) einig. Einige Autoren beziehen sich in ihren Publikationen auf die Art und Weise, wie eine Dokumentation einer API den Nutzern und Entwicklern zur Verfügung gestellt werden soll.

Diese Tabelle listet jene Autoren, welche in ihren Werken eine bestimmte Art der Dokumentation vorstellen.

Hunter [40]
Sahni [43]

Tabelle 10: Dokumentation in der Literatur

Neben der Dokumentation des Quellcodes der API (falls dieser mit angeboten wird), sollte auch eine vollständige Dokumentation für die Entwickler und Nutzer bereitstehen. Für diese Dokumentation geben die Autoren folgende Gestaltungsvorschläge an:

- Die Dokumentation der API sollte für jeden öffentlich zugänglich sein und nicht nur für Nutzer der API. Auch sollte die Dokumentation einfach und schnell zu finden sein. Eine praktikable Möglichkeit dafür, wäre die Dokumentation über eine Website der API anzubieten, ohne das sich ein Nutzer dafür anmelden muss.

- Die Dokumentation sollte in einer druckbaren Version zur Verfügung gestellt werden. Auch wenn physische Ausdrucke eher selten sein werden, wird doch ein Teil der Nutzer die Dokumentation als PDF lokal speichern wollen um auch offline darauf zugreifen zu können.

- Es ist auch sinnvoll, nicht an Beispielen zu sparen und diese auch vollständig aufzuführen. Beispielanfragen und Antworten sollten dementsprechend nicht abgekürzt werden. Diese Beispiele sollten durch Syntaxhighlighting hervorgehoben werden und am besten auch kopierbare Beispiele sein, welche einfach getestet werden können.

- Eine gute Möglichkeit, die Dokumentation zu verbessern, ist das Angebot einer Entwickler-API-Konsole. Dadurch haben Nutzer die Möglichkeit mit der API zu experimentieren.

- Jegliche Änderung in der API muss eiligst in der Dokumentation vermerkt werden, und die Nutzer der API und ihrer Dokumentation müssen auf diese Änderungen hingewiesen werden. Möglichkeiten hierzu wären Mailinglisten und Blogs über die Änderungen.

Auch wenn diese Punkte nicht für die Funktionalität der API notwendig sind, sind sie dennoch besonders wichtig bei der Weiterentwicklung der API und sollten demnach befolgt werden.

4.7.1 Diskussion und Bewertung

Als Softwareentwickler sind wir der Meinung, dass Dokumentation ein essentieller Teil einer jeglichen Software ist. Dementsprechend ist es wichtig, dass Dokumentationen immer einen gewissen Standard erreichen, damit mit ihnen auch gut gearbeitet werden kann. Auch wenn eine saubere Dokumentation nicht notwendig ist, dass eine API RESTful wird, ist es in unseren Augen dennoch notwendig die Vorschläge der Autoren bei dem Design und Entwurf einer REST API zu befolgen.

4.8 Wie soll ich die Fehlerbehandlung entwerfen?

In jeder Software muss sich der Entwickler Gedanken darüber machen, wie er mit Fehlern jeder Art umgehen soll, seien diese nun einem fehlerhaften Entwurf der Schnittstelle, dem Fehlverhalten des Clients oder Änderungen auf dem Server zu Last gelegt. Für RESTful APIs bieten sich den Entwicklern doch noch spezielle Arten den Clients diese Fehler mitzuteilen, bedingt durch das Hypertext Transfer Protocol (HTTP). Dieses liefert durch dessen Statuscodes auch Möglichkeiten, Fehlermeldungen zu übertragen, welche durch ihren Code schon eine Bedeutung besitzen.

Diese nachfolgende Tabelle enthält jene Autoren, welche in ihren Werken eine bestimmte Art der Fehlerbehandlung vorstellen.

Allamaraju [11]
Mulloy [15]
Mulloy [18]
Dhall [21]
Jauker [36]
Sahni [43]

Tabelle 11: Fehlerbehandlung in der Literatur

Obwohl es viele passende HTTP-Statuscodes gibt, ist es doch unpraktisch, alle zu verwenden, da den meisten Nutzern und Entwicklern nicht alle bekannt sind (mit einer `418 - I'm a teapot`-Fehlermeldung ist selten geholfen). Auch tritt durch die Verwendung von Statuscodes, welche nicht `200 - OK` sind, Probleme mit der Verwendung von `Adobe Flash` auf [15]. Dementsprechend sollten die verwendeten Statuscodes auf ein sinnvolles Maß begrenzt werden. Grundlegend sollten diese Fälle abgedeckt sein:

- Alles funktioniert
- Fehler in der Anwendung
- Fehler in der Schnittstelle

Durch die Abdeckung dieser Fälle, sowie einiger weiteren Feinheiten, sollten die HTTP-Statuscodes abgedeckt sein.

Um die Problematik von `Adobe Flash` zum umgehen, präsentiert `Twitter` [6] eine gute Lösung. Durch die Verwendung des optionalen Parameters `suppress_response_codes=true` wird immer `200` als Statuscode zurückgegeben. Den wirklichen Fehlercode sowie dessen Beschreibung befindet sich dann im Körper der `200`-Antwort. Dies könnte dann so aussehen:

```
Anfrage:
GET example.com/fs/197?suppress_response_codes=true

Antwort:
HTTP/1.1 200 OK
{response_code" : "404", "message" : "Ressource
not found."
"more_info" : "http://dev.example.com/errors/12345",
"code" : 12345}
```

Bei der Angabe von Inhalten zur Fehlerbeschreibung sollte neben dem Fehlercode noch eine verbale Beschreibung beigelegt werden, falls ein Entwickler diesen Fehler debuggen möchte. Weitere zusätzliche Informationen zu diesem Fehler und mögliche Beseitigungshinweise können im Nachrichtenkörper angebracht werden. Die Schnittstellen von `Facebook` [7], `Twilio` [8] und `SimpleGeo` [10] bieten dabei folgende Fehlercodes an:

```
Facebook
HTTP Status Code: 200
{"type":"OAuthException","message":"(#803) Some
of the aliases you requesteed do not exist:
foo.bar"}

Twilio
HTTP Status Code: 401
{"status":401,"message":"Authenticate","code":20003,
"more_info":"http//www.twilio.com/docs/errors/20003"}

SimpleGeo
HTTP Status Code: 401
{"code":401,"message":"Authentication Required"}
```

Hierbei sticht besonders `Twilio` hervor, welches neben dem Statuscode noch eine Nachricht (welche aber auch ausführlicher sein dürfte) auch eine Verlinkung auf den Fehler in der Dokumentation enthält. `Facebook` benutzt nur den Statuscode `200` womit die `Adobe Flash`-Problematik umgangen wird, der Fehler aber leichter untergehen kann. Auch ist nicht klar, was der Fehler `803` sein soll. `SimpleGeo` enthält sich jeglicher weiteren Informationen und gibt nur den Statuscode, sowie dessen Bedeutung zurück.

4.8.1 Diskussion und Bewertung

Obwohl durch eine einfache Fehlerbehandlung eine Schnittstelle dennoch RESTful sein kann, sind wir als aus softwaretechnischen Gründen der Meinung, dass durch die Verwendung von `HTTP-Statuscodes` und einer textuellen Beschreibung im Nachrichtenkörper eine Schnittstelle deutlich verbessert werden kann, da so Entwickler die Möglichkeit erhalten, mit ihren Fehlern zu arbeiten und so einfach die Schnittstelle leicht debuggen können. Die Problematik von `Adobe Flash` bezüglich des `200`-Statuscodes, ist in unseren Augen kein Grund die Statuscodes der Antworten zu verändern, sondern eher ein Grund `Adobe Flash` für RESTful APIs nicht zu verwenden.

4.9 Wie biete ich unterschiedliche Sprachen einer Ressource an?

Die durch eine Ressource abrufbaren Inhalte sind oft nicht ausschließlich für Maschinen gedacht, sondern enthalten Elemente, die von Menschen gelesen werden müssen. Besonders bei einer großen, internationalen Zielgruppe tritt daher schnell das Problem der (natürlichen) Sprache auf. Es muss eine Möglichkeit geben, je nach Anfrage den Inhalt in einer anderen Sprache zu erhalten. Mögliche Einsatzszenarien gibt es vor allem bei Webpräsenzen wie Online-Shops oder auch Diensten, welche einer großen Gemeinschaft zugänglich sein sollen (etwa geteilte Kalender).

Ähnlich wie bei der Versionierung bieten sich zwei Hauptlösungen an: Die Sprache wird über die Anfrage-Header bestimmt oder die angefragte URL enthält die Information. Nachfolgende Tabelle bietet einen Überblick darüber, welche Autoren in ihren Veröffentlichungen die Sprachangabe über den Header oder über die URL empfehlen.

Header	URL
Allamaraju [11]	Allamaraju [11]
W3 [23]	cloud.google.com [26]
W3 [44]	
Orabig [31]	
Klabnik [38]	

Tabelle 12: Sprachangabe in der Literatur

4.9.1 Lösungsansatz 1: Sprachangabe im Header

Möchte man beispielsweise die Benutzeroberfläche seines Online-Shops in unterschiedlichen Sprachen anbieten, so handelt es sich bei unterschiedlichen Sprachen noch immer um dieselbe Ressource. Lediglich die Darstellung hat sich geändert. Bei diesem Lösungsansatz werden natürliche Sprachen also als unterschiedliche Repräsentationen derselben Ressource aufgefasst. Die Anfrage einer solche Repräsentation geschieht über die Anfrage-Header. HTTP bietet für diesen Use-Case bereits einen fest definierten Header, das `Accept-Language`-Headerfeld.
Das folgende Beispiel zeigt die grundsätzliche Verwendung. Es wird hierbei eine englische Repräsentation der Online-Shop-Ressource angefragt. In der Antwort wird die zurückgegebene Sprache idealerweise durch den Antwort-Header `Content-Language` angegeben. Fehlt dieser, impliziert dies entweder Unkenntnis über die Sprache der gelieferten Ressource [23] oder zeigt an, dass die Antwort für alle Leser geeignet ist.

```
Anfrage:
GET http://example.org/shop
Accept-Language: en
Antwort:
HTTP/1.1 200 OK
Content-Language: en

<hier folgt der eigentliche Inhalt auf Englisch>
```

Der Anfrage-Header `Accept-Language` kann auch mehrere Sprachen enthalten. Im Folgenden Beispiel wird wieder der Shop angefragt, wobei die drei Sprachen Englisch, Französisch und Deutsch akzeptiert werden:

```
Anfrage:
GET http://example.org/shop
Accept-Language: en, fr, de
Antwort:
HTTP/1.1 200 OK
Content-Language: en

<hier folgt der eigentliche Inhalt auf Englisch>
```

Das obenstehende Beispiel kann ferner dahingehend erweitert werden, dass man Prioritäten für die Sprachen vergibt. Standardmäßig haben die nach dem `Accept-Language`-Feld genannten Sprachen (hier "en", "fr", "de") alle dieselbe Priorität; die Reihenfolge ihrer Nennung spielt also keine Rolle. Die Priorität wird mit einer Zahl angegeben. Je höher, desto stärker wird diese Sprache bevorzugt. Die Zahl kann Werte zwischen 0 und 1 annehmen. Ist kein Wert angegeben, wird 1 angenommen. Der jeweilige Wert wird an die entsprechende Sprache angehängt, sodass sich diese Form ergibt:

```
Accept-Language: <Sprachkürzel>;q=<Priorität>
```

Folgendes Beispiel illustriert diese Priorisierung: Der Leser bevorzugt Texte auf Französisch. Ferner spricht er Englisch und Deutsch. Wenn möglich, möchte er Texte in Britischem Englisch.

```
Anfrage:
GET http://example.org/shop
Accept-Language:fr,en-gb;q=0.8,en;q=0.7,de;q=0.5
Antwort:
HTTP/1.1 200 OK
Content-Language: en

<hier folgt der eigentliche Inhalt auf Englisch>
```

Nicht nur der Client kann mehrere (gewünschte) Sprachen spezifizieren. Auch das Antwort-Feld `Content-Language` kann mehrere Einträge aufweisen. Dies ist für Inhalte für verschiedensprachige Empfänger gedacht, wobei diese Inhalte jedoch mehrere Sprachen beinhalten. Ein Beispiel könnte etwa der Text der Verfassung Quebecs sein: Der Text liegt etwa einmal in Englisch und einmal in Französisch vor, wobei es keine Trennung gibt. Die Antwort des Servers würde in diesem Falle wie folgt lauten:

```
HTTP/1.1 200 OK
Content-Language: en, fr

<hier folgt der eigentliche Inhalt auf Englisch
und Französisch>
```

Die Verwendung mehrerer Sprachen im Antwort-Header ist nicht für solche Inhalte gedacht, die zwar multilingual sind, sich jedoch an gleichsprachige Emfänger richten. Ein Lerndokument eines Französischkurses, welcher sich an Engländer richtet, würde zwar grundsätzlich auch in das obige Beispiel passen, widerspricht jedoch der Intention hinter dem Header-Feld. Korrekt wäre in diesem Fall die Verwendung von nur einer Sprache, nämlich Englisch.

4.9.2 Lösungsansatz 2: Sprachangabe in der URL

Eine andere Möglichkeit, seinen Inhalt in verschiedenen Sprachen anzubieten, besteht im Verändern der URL. Hierbei wird also streng genommen keine andere Repräsentation derselben Ressource verlangt, sondern eine gänzlich andere Ressource angesprochen. Meist wird hierzu das entsprechende Sprachkürzel in die URL eingebaut. Hierbei gibt es zwei verbreitete Möglichkeiten:

- Am Anfang der URL/als Subdomain:
  Das Kürzel wird allem Weiteren vorangestellt. Ein bekanntes Beispiel ist die Wikipedia:
  Deutsche Sprachversion: http://de.wikipedia.org/wiki/Representational_State_Transfer
  Englische Sprachversion: http://en.wikipedia.org/wiki/Representational_state_Transfer
- An anderer Stelle, meist nach der Top-Level-Domain (tld).
  Hierbei taucht das Kürzel nach dem vollständigen Domainnamen auf. Etwa:
  `http://www.example.com/en/shop` oder andere Varianten, z.B.:
  `http://www.example.com/shop/en`
- Im URL-Query als Parameter:
  Hier wird das Sprachkürzel als Parameter übergeben. Meist werden Parameternamen wie `"lang"` oder `"locale"` genutzt. Ein bekanntes Beispiel:
  `http://www.google.de/?hl=en`

4.9.3 Lösungsansatz 3: Keine Angabe der Sprache in der Anfrage und Antwort

Dies ist die einfachste Variante: Hierbei wird der Inhalt in allen verfügbaren Sprachen wiedergegeben. Diese Lösung bietet sich weniger bei von Menschen zu lesenden Inhalten an, da hier potentiell mehr Daten als nötig übertragen werden. Die Filterung nach der "richtigen" Sprache erfolgt in einer Nachbehandlung auf der Client-Seite. Wenn überhaupt, so ist dieser Lösungsansatz nur bei kleineren Datenmengen zu empfehlen. Ein Beispiel wäre etwa die Anfrage zu einem Buch. Als Antwort werden dann sämtliche vorhandene Übersetzungen des Titels geliefert, obwohl später nur eine verwendet wird. Die Antwort wird in JSON gegeben.

```
Anfrage:
GET http://example.org/books/421337
Antwort:
HTTP/1.1 200 OK

{
"id":421337,
"title":[
{"lang":"en", "translation":"Gone with the
Wind"},
{"lang":"de", "translation":"Vom Winde verweht"},
{"lang":"fr", "translation":"Autant en emporte le
vent"}],
"Author":"Margaret Mitchell"
}
```

4.9.4 Diskussion und Bewertung

Wie schon bei der Versionierung sprechen einige Aspekte gegen die Angabe der Sprache in der URL. Unter anderem geht die Verwendung der URL zu Lasten der Wartbarkeit der Schnittstelle: Möchte man etwa die Struktur umstellen, so wird es durch das Beachten der Sprachkürzel nur unnötig kompliziert. Auch wird das Protokoll missachtet: HTTP gibt eine ganz klare Lösungsstrategie für solche Fälle vor: Die Verwendung des Headers. Hier wurden viele weitere Möglichkeiten bedacht - wie etwa die Priorisierung - welche ein Maximum an Flexibilität erlauben. Insbesondere bei Webseiten spricht für die Verwendung der Header auch die leichte Konfigurierbarkeit verschiedenster Browser, sodass immer der Wunsch des Nutzers korrekt abgebildet werden kann. All dies zusammengenommen gibt es keinen Grund, den `Accept-Language`-Header nicht zu verwenden.

5 RESTful Praxisbeispiel: GitHub API

In diesem Kapitel wird anhand einer existierenden und REST-Schnittstelle gezeigt, wie die in Kapitel 4 erläuterten Musterlösungen und Best Practice real und praktikabel umgesetzt werden können. Hierbei wurde die Schnittstelle der Plattform `GitHub` [9] verwendet, da diese von vielen Autoren als besonders RESTful bezeichnet wird und viele Musterlösungen und Best Practices auch umsetzt.

Dabei werden nun, für die in Kapitel 4 gelisteten Musterlösungen, die Beispiele aus der `GitHub`-API aufgeführt (sofern in der API vorhanden).

5.1 Versionierung

Standardmäßig werden von `Github` die Ressourcen der aktuellen Version (zum Zeitpunkt der Erstellung dieser Fachstudie, war dies die Version `v3`) zurückgegeben. Wenn eine bestimmte Version erfragt werden möchte, muss dies über den `Accept-Header` geschehen. Dies würde wie folgt aussehen:

```
Accept: application/vnd.github.v3+json
```

5.2 Ressourcenbenennung

Bei der Benennung der Ressourcen unterscheidet die Schnittstelle zwischen einzelnen Ressourcen, welche namentlich im Singular sind, und Listen von Ressourcen, welche namentlich den Plural benutzen. Der Zugriff auf eine Liste aller Repositorien der Organisation `example` sieht so aus

```
GET /orgs/example/repos
```

und liefert eine Zusammenfassung aller Repositorien. Der Zugriff auf ein bestimmtest Repositorium würde so aussehen

```
GET /repos/example/example.rb
```

5.3 Repräsentationen der Ressourcen

Bezüglich der Angabe von verschiedenen Repräsentationen für Ressourcen bietet `GitHub` für sämtliche Anfragen und Antworten nur `JSON` an und umgeht damit Problematik, dass die Schnittstelle mit bestimmten Formaten nicht arbeiten kann. Des Weiteren wird die Art der Repräsentation im `Content`-Header angegeben.

```
Content-Type: application/json; charset=utf-8
```

5.4 URL-Design

Als Einstiegspunkt (root endpoint) bietet die Schnittstelle zwei URLs an. Von diesen zwei URLs ist eine für die Enterprise-Version gedacht, die andere für jeglichen anderen Nutzer. Die URL der Enterprise-Version lautet

```
yourdomain.com/api/v3
```

während andere Nutzer über die intuitive URL

```
api.github.com
```

auf den Schnittstellenendpunkt kommen können, und von dort aus weiter navigieren.

5.5 API-Effizienz

Um die Kommunikation mit der API effizienter zu machen bietet auch `GitHub` verschiedene Möglichkeiten an. Diese Möglichkeiten sind hauptsächlich `Pagination, Rate Limiting` und der Einsatz von `E-Tags/Last-Modified`.

- **Pagination**
  Die Paginierung wird über die Parameter `?page` und `?per_page` realisiert. Als Standardwert werden 30 Einträge pro Seite angezeigt. Eine Anfrage, beginnend bei Seite 28 und dann die nächsten 100 Einträge, würde so aussehen:

  ```
  $ curl 'https://api.github.com/user/repos?page=28&per_page=100'
  ```

  Das Weglassen des `?page`-Parameters führt dazu, dass die erste Seite zurückgegeben wird.

  Die Paginierung verwendet auch die `Link-Header` von HTTP um automatisch Links auf die nächsten Elemente der nächsten Seite zu bilden. Es wird davon abgeraten diese Links selber zu erstellen. Ein Beispiel dafür ist in Abschnitt 4 im Unterabschnitt `Wie mache ich meine Schnittstelle effizienter?` zu finden oder in der Dokumentation der Schnittstelle, welche auch eine Tabelle mit den möglichen Werten für den `Location-Header` enthält.

- **Rate Limiting**
  Die `GitHub`-Schnittstelle ermöglicht es, die Anfrage auf Ressourcen so zu begrenzen, dass nur eine bestimmte Anzahl von Anfragen pro Stunde gestellt werden kann. Dabei kann zwischen authentifizierten und nicht-authentifizierten Benutzern in der Anzahl der Anfragen unterschieden werden. Dadurch können Angriffe auf die Verfügbarkeit der Ressourcen unterbunden werden. Dafür werden die HTTP-Header `X-RateLimit-Limit, X-RateLimit-Remaining` und `X-RateLimit-Reset` verwendet. Eine genauere Beschreibung findet sind in der Dokumentation der Schnittstelle.

- **ETags/Last-Modified**
  Um zu überprüfen, ob sich eine Ressource seit der letzten Anfrage geändert hat, stellt die API die Header `ETag` und `Last-Modified`, sowie deren weiteren Methoden und Header zur Verfügung.

5.6 Hypermedia

Es ist allen Ressourcen möglich, mit dem `*_url`-Feld von HTTP ausgestattet zu werden und somit auf andere Ressourcen zu verweisen. Dadurch werden die URLs von der Schnittstelle erstellt, und dies muss nicht vom Client händisch ausgeführt werden. Dadurch können auch zukünftige Aktualisierungen einfacher entwickelt werden.

5.7 Dokumentation

Über die Website (`https://developer.github.com/v3/`) der Schnittstelle ist deren Dokumentation öffentlich verfügbar. Ebenfalls werden Beispiele für die Nutzung und Handhabung der Schnittstelle aufgeführt, welche in einem kopierbaren Format sind, damit Entwickler mit ihnen Arbeiten können.

5.8 Fehlerbehandlung

`GitHub` stellt drei mögliche Fehlermeldungen für Client Errors zur Verfügung.

- Auf eine Anfrage mit ungültigem `JSON` erhält man eine `400 Bad Request` Antwort.

  ```
  HTTP/1.1 400 Bad Request
  Content-Length: 35

  {"message":"Problems parsing JSON"}
  ```

- Auch bei einer Anfrage mit dem falschen Typ von JSON-Werten wird mit einem 400 Bad Request geantwortet.

```
HTTP/1.1 400 Bad Request
Content-Length: 40

{"message":"Body should be a JSON Hash"}
```

- Und bei einer Anfrage mit ungültigen Feldern bekommt der Client ein 422 Unprocessable Entity zurück.

```
 HTTP/1.1 422 Unprocessable Entity
Content-Length: 149

{"message": "Validation Failed","errors":
[{"resource": "Issue", "field": "title", "code":
"missing_field"}]}
```

Auch haben alle Fehlerobjekte die Möglichkeit über Felder und Fehlercodes mitzuteilen, weshalb die Anfrage nicht bearbeitet werden konnte. Die Dokumentation bietet hierbei eine passende Tabelle.

5.9 Weitere Möglichkeiten der API

Die Schnittstelle bietet noch weitere Optionen, die ihre Nutzer verwenden können. Auch wenn diese Optionen manchmal nicht direkt für eine REST-Schnittstelle notwendig sind, zählen wir sie hier nun auf, da sie Elemente enthalten, welche in Unterabschnitten von Abschnitt 4 aufgeführt werden.

5.9.1 Nutzung der HTTP-Verben

Von den HTTP-Verben sind in der GitHub-API die Verben HEAD, GET, POST, PATCH, PUT und DELETE implementiert. Eine Beschreibung dessen, was diese Verben genau machen, und inwiefern sie an die Bedürfnisse der Schnittstelle angepasst wurden, ist als Tabelle in der GitHub-Dokumentation einsehbar.

5.9.2 HTTP Weiterleitungen (HTTP Redirects)

Die Schnittstelle von GitHub ist in der Lage HTTP-Weiterleitungen zu verarbeiten. Dementsprechend müssen die Clients davon ausgehen, auf eine Anfrage eine Weiterleitung zu erhalten. Die API benutzt hierfür den HTTP Location-Header und sendet dem Client die URI der Ressource zurück, an welche er die Anfrage stellen soll.

5.9.3 Parameter

Für viele Methoden der Schnittstelle sind optionale Paramter möglich. Dabeie wird zwischen GET und den anderen Methoden (POST, PATCH, PUT, DELETE unterschieden. Für GET-Anfragen gilt, dass alle Parameter, welche nicht schon im Pfad spezifiziert sind, als Anfrageparameter (query string parameter) übergeben werden können.

```
https://api.github.com/repos/vmg/red/issues?state=open
```

wobei vmg und red Werte sind, welche für den Besitzer und das Repositorium stehen und schon im Pfad angegeben werden, während state über den Anfrageparamter übergeben wird.

Für die anderen Methoden gilt, dass alle Parameter, die nicht im Pfad angegeben werden, als JSON mit einem Content-Header mitgegeben werden müssen.

6 Zusammenfassung und Ausblick

In dieser Fachstudie wurden Musterlösungen und Best Practices für das Entwurf und die Realisierungen von REST-Schnittstellen aufgeführt und miteinander verglichen. Es wurden die Grundbegriffe und Prinzipien für RESTful APIs erläutert, sowie das Hypertext Transfer Protocol (HTTP) als Grundprotokoll für REST-Schnittstellen dargelegt. Die häufigsten Musterlösungen für den Entwurf von REST-APIs wurden aufgezählt und erklärt. Dabei wurden verschieden Implementierungen vorgestellt und miteinander verglichen. Dies geschah, indem die Vor- und Nachteile der jeweiligen Implementierungsmethoden einander gegenüber gestellt wurden und die Meinung der Autoren, auf deren Werken diese Fachstudie basiert, der von ihnen präferierten Implementierung hinzugefügt wurde. Dazu wurden von den Autoren dieser Fachstudie noch eine eigene Bewertung abgegeben, welche mögliche Implementierung in ihren Augen die RESTfulste ist. Als praktisches Beispiel wurde die Schnittstelle von `GitHub` betrachtet, welche von vielen Autoren als ein Musterbeispiel für eine RESTful API beschrieben wird. Hierbei wird betrachtet, inwiefern die möglichen Musterlösungen in der Schnittstelle von `GitHub` realisiert wurden und diese mit Beispielen aus der Dokumentation der `Github`-Schnittstellen belegt.

Noch werden viele Schnittstellen welche über HTTP kommunizieren als RESTful bezeichnet, obwohl sie die von Roy Fielding definierten Grundprinzipien für REST nicht erfüllen. Dennoch erstehen in den letzten Jahren immer mehr Blogs und andere Onlinedokumente, welche sich mit den Musterlösungen und Best Practices für REST-Schnittstellen befassen. Allerdings bauen diese Musterlösungen alle auf HTTP auf, obwohl dieses Protokoll für REST kein Kriterium ist. Auch sind nicht alle Musterlösungen mit der Spezifikation von HTTP immer konform und die Grundprinzipien von REST werden auch auf ihre eigene Art ausgelegt.

Um eine völlig konforme Schnittstelle zur Definition von REST zu haben, muss also noch einiges getan werden, auch wenn die Autoren Änderungen und Abweichungen mit dem praktischen Aspekt ihrer Lösungen begründen.

Literatur

[1] http://en.wikipedia.org/wiki/HATEOAS.

[2] http://en.wikipedia.org/wiki/Uniform_resource_locator.

[3] http://de.wikipedia.org/wiki/Idempotenz.

[4] http://de.wikipedia.org/wiki/Representational_State_Transfer.

[5] http://de.wikipedia.org/wiki/Hypermedia.

[6] https://dev.twitter.com/.

[7] https://developers.facebook.com/.

[8] http://www.twilio.com/docs/api/rest.

[9] GitHub API. https://developer.github.com/v3/.

[10] SimpleGEO. https://github.com/simplegeo.

[11] Subbu Allamaraju. *RESTful Web Services Cookbook*. O'REILLY, YAHOO! PRESS, 2010. 1. Auflage.

[12] Brian Mulloy. RESTful API Design: can your API give developers just the information they need? https://blog.apigee.com/detail/restful_api_design_can_your_api_give_developers_just_the_information, 2011.

[13] Brian Mulloy. RESTful API Design: nouns are good, verbs are bad. https://blog.apigee.com/detail/restful_api_design_nouns_are_good_verbs_are_bad, 2011.

[14] Brian Mulloy. RESTful API Design: tips for versioning. https://blog.apigee.com/detail/restful_api_design_tips_for_versioning/, 2011.

[15] Brian Mulloy. RESTful API Design: tips for handling exceptional behavior. https://blog.apigee.com/detail/restful_api_design_tips_for_handling_exceptional_behavior, 2012.

[16] Brian Mulloy. API Design: Harnessing HATEOAS, Part 1. https://blog.apigee.com/detail/api_design_harnessing_hateoas_part_1, 2013.

[17] Brian Mulloy. API Design: Harnessing HATEOAS, Part 2. https://blog.apigee.com/detail/api_design_harnessing_hateoas_part_2, 2013.

[18] Brian Mulloy. RESTful API Design: what about errors? https://blog.apigee.com/detail/restful_api_design_what_about_errors/, 2013.

[19] Brian Mulloy and Kevin Swiber. API Design: Binary Data, Caching, Transactions, and More. https://blog.apigee.com/detail/Binary_Data_Caching_Transactions_and_More, 2013.

[20] Chander Dhall. 5 Best Practices for Better RESTful API Development. http://devproconnections.com/web-development/restful-api-development-best-practices, 2013.

[21] Chander Dhall. 5 More Tips to Improve Your RESTful API Development. http://devproconnections.com/web-development/5-more-tips-restful-api-development, 2013.

[22] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. Master's thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[23] Fielding et al. RFC 2616. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.

[24] Freed and Klensin, RFC 4288. RFC 4288. http://tools.ietf.org/html/rfc4288#section-3.2.

[25] Geert Jansen. Thoughts on RESTful API Design. https://restful-api-design.readthedocs.org/en/latest/.

[26] Google. Using REST. https://cloud.google.com/translate/v2/using_rest#language-params.

[27] Gregor Roth. RESTful HTTP in practice . http://www.infoq.com/articles/designing-restful-http-apps-roth.

[28] Savas Parastatidis Jim Webber and Ian Robinson. *REST in Practice - Hypermedia and Systems Architecture*. O'REILLY, 2010. 1. Auflage.

[29] Martin Fowler. Richardson Maturity Model. http://martinfowler.com/articles/richardsonMaturityModel.html, 2010.

[30] Mark Massé. *REST API - Design Rulebook*. O'REILLY, 2011. 1. Auflage.

[31] Orabîg@stackoverflow. Supporting multiple languages in a REST API. http://stackoverflow.com/questions/25353854/supporting-multiple-languages-in-a-rest-api.

[32] Peter Williams. Versioning REST Web Services. http://barelyenough.org/blog/2008/05/versioning-rest-web-services/.

[33] Leonard Richardson and Sam Ruby. *RESTful Webservices*. O'REILLY, 2007. 1. Auflage.

[34] Roy Fielding. REST APIs must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008.

[35] Scott Seely. Versioning REST Services. http://www.informit.com/articles/article.aspx?p=1566460.

[36] Stefan Jauker. 10 Best Practices for Better RESTful API. http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/, 2014.

[37] Steve Klabnik. Haters gonna HATEOAS. http://timelessrepo.com/haters-gonna-hateoas.

[38] Steve Klabnik. Nobody Understands REST or HTTP. http://blog.steveklabnik.com/posts/2011-07-03-nobody-understands-rest-or-http.

[39] Cesare Pautasso Thomas Erl, Benjamin Carlyle and Raj Balasubramanian. *SOA with REST - Principles, Pattern and Constraints for Building Enterprise Solutions with REST*. PRENTICE HALL, 2012. 1. Auflage.

[40] Thomas Hunter II. Principles of good RESTful API Design. http://codeplanet.io/principles-good-restful-api-design/, 2013.

[41] Stefan Tilkov. *REST und HTTP - Einsatz der Architekur des Web für Integrationsszenarien*. dpunkt.verlag, 2011. 2. Auflage.

[42] Todd Fredrich. RESTful Service Best Practices - Recommendations for Creating Web Services. http://www.RestApiTutorial.com, 2013.

[43] Vinay Sahni. Best Practices for Designing a Pragmatic RESTful API. http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#docs, 2013.

[44] W3.org. Accept-Language used for locale setting. http://www.w3.org/International/questions/qa-accept-lang-locales.en.

Alle Links wurden zuletzt am 01.10.2014 abgerufen und auf ihre Gültigkeit überprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, den 01.10.2014

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

Stuttgart, 01.10.2014