RESTful Web-Services mit Qualität: Teil 1: Mit Best Practices zu einem qualitätsorientierten Entwurf

Der Trend, Web-Services auf der Basis von REST umzusetzen, ist ungebrochen. Die steigende Popularität von REST schlägt sich auch in der Anzahl an Frameworks nieder, die den Einstieg in die Entwicklung von RESTful Web-Services zusätzlich vereinfachen. Die zunehmende Verbreitung von REST und der damit verbundene Einfluss auf die Qualität einer gesamten Architektur erfordern es daher, die Qualität von RESTful Web-Services stärker als bisher zu betrachten. Dieser Artikel fasst Best Practices für RESTful Web-Services zusammen, die berücksichtigt werden sollten, um Anforderungen an eine qualitativ hochwertige Architektur gerecht zu werden.

Besteht in einem Projekt die Anforderung, Web-Services zu entwickeln, so fällt die Entscheidung heutzutage häufig auf REST als Paradigma (vgl. [Inf11]). Meist sind die Gründe hierfür, dass REST im Vergleich zu alternativen Ansätzen wie Web-Services auf Basis von SOAP eine geringere Komplexität aufweist und eine leichtgewichtigere Übertragung von Daten ermöglicht. Zusätzlich ist die große Funktionsvielfalt von SOAP in nur wenigen Projekten tatsächlich erforderlich.

Die zunehmende Verwendung von REST in Projekten bedeutet jedoch, dass der Einfluss von REST auf die gesamte Architektur stark zugenommen hat. Es gilt daher mehr denn je, die Gestaltung von Web-Services auf Basis von REST mit besonderer Sorgfalt durchzuführen und dem qualitätsorientierten Entwurf von RESTful Web-Services eine wesentliche Rolle zukommen zu lassen, um eine hohe Qualität der gesamten Architektur sicherzustellen.

Aufgrund des großen Gestaltungsspielraums beim Entwurf von RESTful Web-Services wurden in den letzten Jahren sowohl seitens der Forschung als auch der Industrie diverse Best Practices erarbeitet, die diesen Gestaltungsspielraum weiter einschränken und Empfehlungen geben, wie ein qualitativ hochwertiger RESTful Web-Service gestaltet werden sollte. Für Architekten und Entwickler ergeben sich hierbei drei wesentliche Herausforderungen:

- Best Practices sind heutzutage in verschiedenen Literaturquellen beschrieben. Das heißt, in einem ersten Schritt müssen diese Quellen gefunden, die jeweilige Beschreibung interpretiert und eine einheitliche Menge an Best Practices abgeleitet werden.
- Gegebenenfalls müssen Best Practices priorisiert werden, da sich diese entweder gegenseitig ausschließen oder aus Kostengründen nicht vollständig berücksichtigt werden können.
- Die Überprüfung, ob die Best Practices eingehalten wurden, erfordert zunächst die Übertragung auf die jeweils genutzte Technologie, wie beispielsweise JAX-RS im Java-Umfeld. Die Überprüfung selbst geht zusätzlich mit einem hohen manuellen Aufwand einher und ist somit aufgrund der Komplexität heutiger Systeme fehlerbehaftet.

Wir werden zeigen, wie Architekten und Entwickler diese Herausforderungen systematisch angehen und RESTful Web-Services in Projekten effizient prüfen können. Hierfür

Bei *REST (REpresentational State Transfer)* handelt es sich um einen Architekturstil, der von Fielding (vgl. [Fie00]) im Rahmen seiner Dissertation entwickelt wurde. Als Grundlage für den Entwurf von REST identifizierte Fielding zunächst vier Schlüsseleigenschaften der WWW-Architektur, die für den Erfolg des Web verantwortlich waren. Um diese Eigenschaften zu erfüllen, leitete er sechs Restriktionen ab, die heute die Grundpfeiler von REST darstellen. Heutzutage wird ein Web-Service, der diese Restriktionen berücksichtigt, als RESTful bezeichnet. Dabei verwaltet ein RESTful Web-Service eine oder mehrere Ressourcen, mit denen über Repräsentationen interagiert werden kann. Die Konzepte und ihre Zusammenhänge sind in **Abbildung 1** dargestellt.

Kasten 1: Die Entstehung von REST.

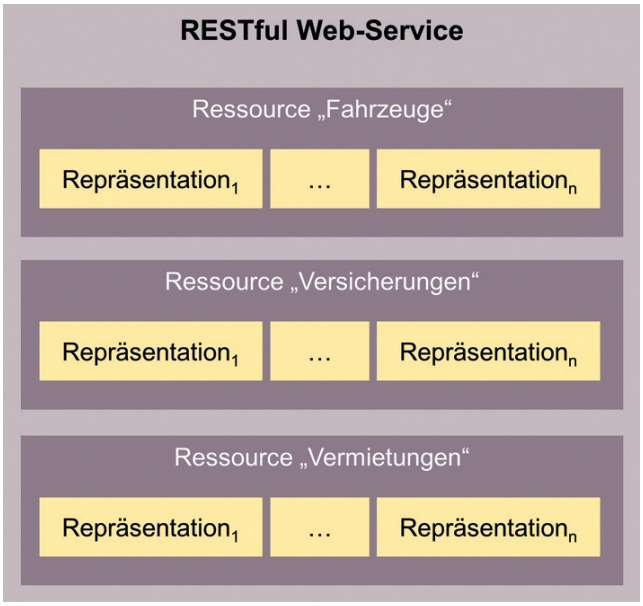

Abb. 1: Konzepte von REST.

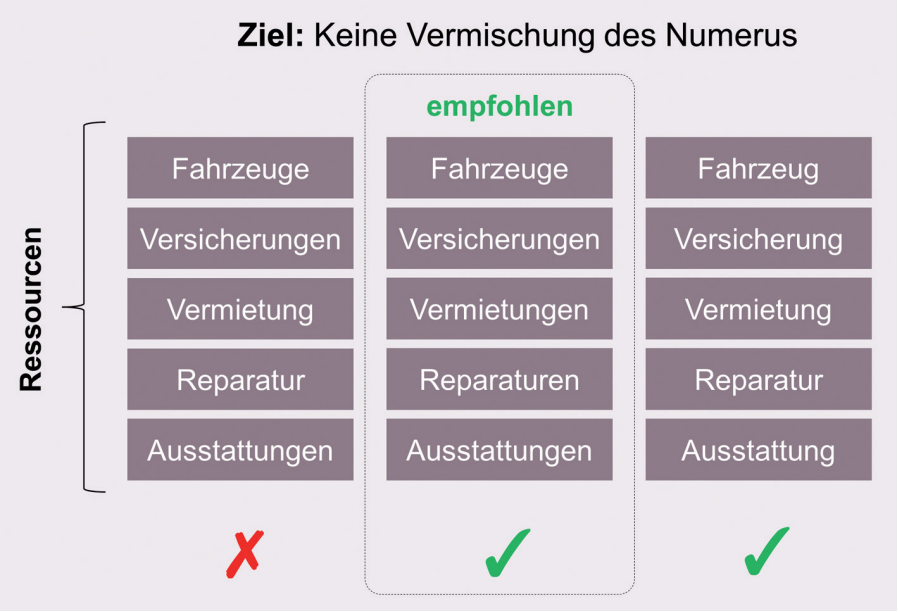

Abb. 2: Keine Vermischung des Numerus bei der Bezeichnung der Ressourcen.

stellen wir im ersten Teil dieses zweiteiligen Artikels eine Menge an Best Practices vor, die für den Entwurf von RESTful Web-Services relevant sind und deren Qualität beeinflussen (*Herausforderung 1*). Die Best Practices stammen dabei aus bestehender Literatur und vereinheitlichen diese.

Um eine Priorisierung der Best Practices zu ermöglichen (*Herausforderung 2*), werden wir im zweiten Teil des Artikels die Best Practices mit Qualitätsmerkmalen wie Wartbarkeit und Effizienz aus der ISO/IEC 25010:2011 (vgl. [ISO]) verknüpfen. Zusätzlich werden wir zeigen, wie auf Basis der Ergebnisse von Michael Gebhart (vgl. [Geb11]) eine Automatisierung durch Werkzeuge erfolgen kann. Um die Ergebnisse in Projekten einzusetzen, stellen wir im zweiten Teil einen Bewertungsbogen zum Download bereit, mit dem RESTful Web-Services in Projekten systematisch geprüft werden können.

Best Practices für REST

Für REST gibt es eine Vielzahl an unterschiedlichen Best Practices, die wir nachfolgend beleuchten wollen. Wir gruppieren hierzu die gesammelten Best Practices in acht verschiedene Kategorien und setzen die Nutzung des Anwendungsprotokolls *Hypertext Transport Protocol (HTTP)* (vgl. [RFC2616]) für die nachfolgende Betrachtung voraus.

Versionierung

Eine Versionierung ist gemäß Brian Mulloy (vgl. [Mul12]) eine der wichtigsten Betrachtungen beim Entwurf eines Web-Service, weshalb dessen Schnittstelle (nachfolgend als *Web-API* bezeichnet) niemals ohne eine Versionserkennung veröffentlicht werden sollte. Dem gegenüber steht die Aussage von Fielding: „Versioning an interface is just a polite way to kill deployed applications“ (vgl. [Fie13]). Durch das Prinzip von *Hypermedia As The Engine Of Application State (HATEOAS)* kann ein RESTful Web-Service mit einer herkömmlichen Web-Seite verglichen werden, wodurch eine Versionierung obsolet wird. Dies setzt natürlich voraus, dass HATEOAS sowohl bei der Entwicklung des Web-Service als auch bei dessen Dienstnehmer berücksichtigt wird.

Beschreibung von Ressourcen

Die Beschreibung von Ressourcen ist für die Benutzbarkeit eines Web-Service entscheidend, da die Ressourcen die Domäne abbilden. Hierfür konnten wir folgende sechs Best Practices identifizieren:

- Für die Bezeichnung von Ressourcen sollten Nomen verwendet werden, da dies eine etablierte Konvention darstellt (vgl. [Mul12], [Vin08], [Pa14]).
- Die Ressourcen sollten domänenspezifisch und konkret benannt sein, sodass deren Bedeutung abgeleitet werden kann (vgl. [Jau14], [Pa14]).
- Die Anzahl der Ressourcen sollte zwischen 12 und 24 liegen, um so die Komplexität des Systems zu begrenzen (vgl. [Mul12]). Dadurch muss unter Umständen ein Kompromiss zwischen der Bezeichnung der Ressourcen und deren Anzahl eingegangen werden.
- Auf die Vermischung des Numerus bei der Bezeichnung von Ressourcen sollte verzichtet werden, während gleichzeitig eine Präferenz für die Pluralform ausgesprochen wird (**siehe Abbildung 2**).
- Vergleichbar mit Klassen bei der Objektorientierung können auch Ressourcen Attribute enthalten. Für deren Bezeichnung sollte die Binnenmajuskel- oder auch Kamelhöcker-Schreibweise mit führendem Kleinbuchstaben verwendet werden, welche sich auch in der Programmierung etabliert hat (vgl. [Mul12]).
- Da der Medientyp *JavaScript Object Notation (JSON)* immer häufiger bei der Interaktion mit Web-Services eingesetzt wird, sollten bei der Bezeichnung der Attribute die Namenskonventionen von JavaScript beachtet werden (vgl. [Mul12]).

Identifizierung von Ressourcen

Die Identifizierung von Ressourcen erfolgt gemäß Fielding (vgl. [Fie00]) mithilfe eines *Uniform Resource Identifier (URI)*. Hierfür konnten wir die folgenden vier Best Practices identifizieren:

- Die URI sollte gemäß der Affordanz selbsterklärend sein (vgl. [Mul12]). Bei der Affordanz handelt es sich um eine Designeigenschaft, der zu Folge ein Gegenstand ohne Erfordernis einer Anleitung benutzt werden kann (vgl. [Mul12]). Da eine URI bei REST in der Regel aus Ressourcen besteht, sprechen wir hier von einer positiven Korrelation mit der konkreten und domänenspezifischen Bezeichnung einer Ressource (siehe Beschreibung von Ressourcen, Punkt 2). Diese These konnte durch die Untersuchung mehrerer bekannter Web-Services, darunter Foursquare und Twitter, belegt werden.
- Eine Ressource sollte lediglich über zwei URIs adressiert sein, wobei die erste eine Ansammlung von Zuständen einer Ressource r und die zweite einen spezifischen Zustand aus dieser An-

```
1 SELECT * FROM r;
2 SELECT * FROM r WHERE uuid = i;
```

Listing 1: Zwei SQL-Abfragen zum Abrufen von Informationen einer Ressource „r“.

sammlung mithilfe eines Identifikators *i* adressiert (vgl. [Mul12]). Dies ist vergleichbar mit einer Datenbankabfrage gemäß **Listing 1**.

- Der Identifikator zur Adressierung eines spezifischen Zustands sollte nur schwer vorhersagbar sein, um so zum Beispiel einen unbeabsichtigten Datenverlust zu verhindern (vgl. [Pap14]).
- Die Verwendung von Verben innerhalb einer URI gilt es zu vermeiden, da dies einen methodenorientierten Ansatz wie z. B. bei SOAP implizieren würde. Hierzu liefert Mulloy ein Beispiel, welche Auswirkungen ein derartiger Ansatz mit sich bringen würde (vgl. [Mul12]).

Fehlerbehandlung

Die Web-API repräsentiert den zentralen Zugangspunkt eines RESTful Web-Service. Jegliche Information zur Implementierung des RESTful Web-Service wird durch diese verborgen, wodurch lediglich das äußere Verhalten betrachtet werden kann. Das äußere Verhalten ergibt sich durch Rückmeldungen des RESTful Web-Service. So müssen bei einem Fehler die resultierenden Fehlermeldungen Aufschluss auf die Fehlerursache geben, da zumeist kein Zugriff auf die Implementierung des RESTful Web-Service existiert und somit die üblichen Techniken zum Debuggen nicht anwendbar sind. Um die Aussagekraft der Fehlermeldungen sicherzustellen, konnten wir drei Best Practices identifizieren:

- Bei auftretenden Fehlern auf Seite des RESTful Web-Service sollten nach Stefan Jauker (vgl. [Jau14]), Steve Vinoski (vgl. [Vin08]) und Patroklos Papapetrou (vgl. [Pap14]) spezifische HTTP-Statuscodes an den Dienstnehmer übermittelt werden.
- Laut der derzeitigen HTTP-Spezifikation der IETF (vgl. [RFC2616]) und [RFC6586]) existieren über 40 verschiedene HTTP-Statuscodes, von denen viele RESTful Web-Services nur eine Teilmenge nutzen. Nach [Mul12] sollten nicht mehr als acht verschiedene HTTP-Statuscodes verwendet werden, was im Widerspruch zur Aussage von Jauker steht: „We don’t need them all, but there should be used at least a mount of 10“ (vgl. [Jau14]). Grundsätzlich ist die Anzahl der HTTP-Statuscodes abhängig vom jeweiligen RESTful Web-Service und daher nicht verallgemeinerbar. Als Richtwert empfehlen wir jedoch 8 bis 15 verschiedene HTTP-Statuscodes zu verwenden, da es sich dabei aus unserer Sicht zum einen um eine handhabbare Menge handelt und zum anderen bei der Untersuchung von sechs verschiedenen RESTful Web-Services sowohl der Durchschnitt als auch der Median bei 12 HTTP-Statuscodes lag.
- Eine ausführliche Fehlerbeschreibung ist für das Verständnis der Fehlerursache von zentraler Bedeutung. Aus diesem Grund sollte sowohl nach [Jau14] als auch nach [Mul12] eine Fehlerbeschreibung aus vier Bestandteilen zusammengesetzt sein: 1) Eine Nachricht für den Entwickler, die die Fehlerursache beschreibt und wenn möglich bereits Hinweise zur Lösung liefert. 2) Eine Nachricht für den Nutzer, die an diesen weitergegeben werden kann. 3) Ein applikationsspezifischer Fehlercode. 4) Ein Hyperlink, der auf weiterführende Informationen verweist (**siehe Listing 2**).

```
1  HTTP/1.1 404 NOT FOUND
2  /* Weitere Header-Informationen */
3  {
4    „error“ : {
5       „errorCode“    : „107“,
6       „messages“     : {
7         „developer“ : „Die angeforderte Ressource ‚userd‘ konnte nicht
                         gefunden werden. Meinten Sie vielleicht ‚users‘ ?“,
8          „user“      : „Bei der Anfrage ist leider ein Fehler aufgetreten.
                        Bitte versuchen Sie es noch einmal zu einem
                        späteren Zeitpunkt.“,
9       },
10       „additionalInfo“ : „.../docs/errors/107“,
11   }
12 }
```

Listing 2: Korrekte Fehlerbehandlung mit JSON.

Dokumentation der RESTful Web-API

Die Dokumentation der Web-API zählt zu den umstrittensten Artefakten im Kontext von RESTful Web-Services, da diese eine zusätzliche Information (*out-of-band information*) darstellt und nach Roy T. Fielding unbedingt zu vermeiden ist: „Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type“ (vgl. [Fie08]). Seine Aussage begründet Fielding damit, dass eher die Dokumentation die Art der Interaktion leiten würde als Hypermedia. So werden beispielsweise Hyperlinks zu Ressourcen entsprechend der Dokumentation oftmals direkt in den Programmcode des Dienstnehmers eingebettet und Geschäftsprozesse dementsprechend umgesetzt. Eine Modifikation der RESTful Web-API führt nun im schlimmsten Fall zu einem nicht funktionierenden Dienstnehmer. Diesen Umstand, bei dem die Dokumentation das zentrale Artefakt bei der Entwicklung eines RESTful Web-Service darstellt, bezeichnen wir als *Documentation As The Engine Of application State (DATEOS)*. Zu beachten ist, dass an dieser Stelle explizit die Dokumentation der Web-API gemeint ist, da lediglich die Web-API für den Endnutzer sichtbar ist, nicht jedoch die Implementierung des Web-Service. Um dies zu vermeiden, benötigen wir für RESTful Web-Services eine neue Art von Dokumentation, die folgende Bestandteile aufweist:

- Einige Beispiele, wie nach dem Prinzip von HATEOAS mit dem RESTful Web-Service zu interagieren ist, da vielen Architekten und Entwicklern der Begriff HATEOAS nicht bekannt ist.

HTTP-Methode	Seiteneffektfreiheit	Idempotent
POST	Nein	Nein
GET	Ja	Ja
PUT	Nein	Ja
DELETE	Nein	Ja

Tabelle 1: Eigenschaften der häufigsten HTTP-Methoden.

- Ein abstraktes Ressourcenmodell in Form eines Zustandsdiagramms, das lediglich die Beziehungen der Ressourcen aufzeigt und nicht deren konkrete URIs (**siehe Abbildung 3**). Damit einhergehend sollte eine semantische Beschreibung der Ressource sowie deren Attribute erfolgen, die sowohl von Maschinen als auch von Menschen gelesen werden kann.
- Ein Nachschlagewerk für auftretende Fehler (vgl. Kategorie „Fehlerbehandlung“).

Verwendung von Parametern

Viele RESTful Web-Services erweitern die URI einer Ressource, um optionale Informationen an einen Web-Service mithilfe von Parametern zu übermitteln (vgl. [Mul12]). Nachfolgend behandeln wir Anwendungsbeispiele zum Filtern, Sortieren, Selektieren und Paginieren von Informationen, da diese von einer Vielzahl an RESTful Web-Services unterstützt werden.

- Zur Filterung von Informationen einer Ressource können entweder deren Attribute oder eine spezielle Abfragesprache verwendet werden (vgl. [Jau14]). Die Wahl ist abhängig von der gewünschten Mächtigkeit der Filterung. So sind einfache Filterungen über die Bezeichnung der Attribute möglich, während für komplexe Filterungen eine Abfragesprache erforderlich ist.
- Für das Sortieren von Informationen empfiehlt Jauker (vgl. [Jau14]) den URI-Parameter **`sort`**, dem eine kommaseparierte Liste von Attributen der Ressource mit jeweils führendem „+“ für eine aufsteigende Sortierung oder „-“ für eine absteigende Sortierung zugewiesen wird. Die Reihenfolge der Attribute repräsentiert die Rangfolge der Sortierung.
- Die Selektion ermöglicht die Auswahl von Attributen einer Ressource, wodurch nur die benötigten Informationen an den Dienstnehmer übermittelt werden. Ähnlich zur Sortierung wird sowohl in [Jau14]) als auch in [Mul12] eine kommaseparierte Liste vorgeschlagen, die dem URI-Parameter **`fields`** zugewiesen wird.
- Die Paginierung ermöglicht die Aufteilung von Informationen auf mehrere virtuelle Seiten, während entsprechende Verweise zum Vor- und Zurückblättern (**`next`** und **`prev`**) sowie zum Springen an Anfang und Ende (**`first`** und **`last`**) existieren. Als URI-Parameter wird **`offset`** und **`limit`** empfohlen, wobei Ersterer die virtuelle Seite identifiziert und Letzterer die Anzahl an Informationen auf der Seite definiert.

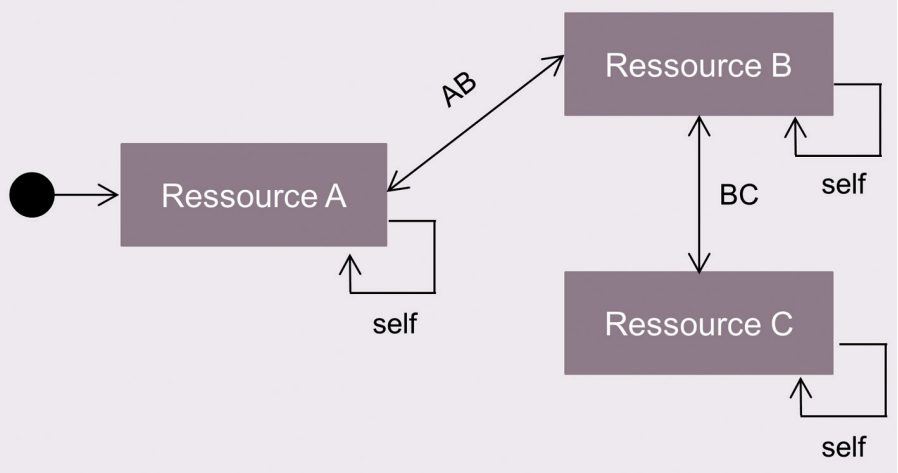

Abb. 3: Abstraktes Ressourcenmodell.

Interaktion mit Ressourcen

Bei REST agiert ein Dienstnehmer niemals direkt mit einer Ressource, sondern nur über deren Repräsentationen. Die konkrete Interaktion erfolgt wiederum mit HTTP-Methoden.

Die HTTP-Methoden sollten nach [Jau14] und [Mul12] entsprechend der gegenwärtigen HTTP-Spezifikation (vgl. [RFC2616]) verwendet werden. So sollte beispielsweise die HTTP-Methode **`GET`** (kurz: **`HTTP-GET`**) nur für seiteneffektfreie und idempotente Operationen benutzt werden. Seiteneffektfreiheit bedeutet, dass diese Operation keinerlei Änderungen am Zustand der Ressource vornimmt. Idempotenz besagt, dass sich der Zustand durch mehrfaches Ausführen der gleichen Funktion nicht verändert. **Tabelle 1** fasst die Eigenschaften für die HTTP-Methoden **`GET`**, **`POST`**, **`PUT`** und **`DELETE`** zusammen.

Im Hinblick auf Operationen zum Erstellen, Lesen, Modifizieren und Löschen (**`CRUD`**) von Informationen müssen diese den in **Tabelle 1** aufgeführten HTTP-Methoden zugewiesen werden. Über die Jahre hat sich die in **Tabelle 2** dargestellte Zuweisung sowohl in der Praxis als auch in der Literatur etabliert (vgl. [Ric10], [Web10]).

Die Unterstützung der HTTP-Methode **`OPTIONS`** (kurz: **`HTTP-OPTIONS`**) wird empfohlen, da die Ressourcen nicht immer alle Operationen zur Verfügung stellen sollen (vgl. [Vin08]). So kann es etwa sein, dass die Ressource „Nutzer“ lediglich einen lesenden Zugriff erlaubt. Dies führt dazu, dass jegliche Schreiboperationen nach Beendigung der Übertragung mit einem entsprechenden HTTP-Statuscode beantwortet werden, der nach RFC2616 dem HTTP-Statuscode „405“ entspricht. Bei Anfragen mit geringen Datenmengen im Inhaltsbereich von HTTP stellt dieses Vorgehen kein Problem dar. Da-

HTTP-Methode	CRUD-Operation	Beschreibung
POST	Erstellen (Create)	Anlegen von neuen Informationen
GET	Lesen (Read)	Abrufen von Informationen
PUT	Modifizieren (Update)	Aktualisieren von Informationen
DELETE	Löschen (Delete)	Löschen von Informationen

Tabelle 2: Zuweisung von CRUD-Operationen zu HTTP-Methoden.

gegen muss bei größeren Datenmengen die Verbindung über eine längere Zeit aufrechterhalten werden, da ein HTTP-Statuscode erst nach Abschluss der Übertragung übermittelt wird. Um dies zu verhindern, kann `HTTP-OPTIONS` verwendet werden, womit erlaubte Methoden im Vorfeld abgerufen werden können.

Die Unterstützung von „conditional GET" sollte nach Vinoski bei der Entwicklung von RESTful Web-Services berücksichtigt werden, um unnötigen Datenverkehr zu verhindern. Mit „conditional GET" werden bereits empfangene Informationen nur bei etwaigen Änderungen erneut an den Dienstnehmer übermittelt. Zu dessen Umsetzung existieren zwei verschiedene Ansätze (vgl. [Vin08]):

- Mithilfe der HTTP-Header-Felder `Last-modified` und `If-modified-since`.
- Mithilfe von so genannten „entity tags".

Unterstützung von MIME-Typen

HTTP nutzt zur Identifizierung von Datenformaten so genannte *MIME*-Typen (*Multipurpose Internet Mail Extension*), die von der IANA registriert und über deren Web-Seite veröffentlicht werden (vgl. [IANA14], [Bin08]). Diese MIME-Typen entsprechen den Repräsentationsformaten bei REST. Nachfolgend konnten wir vier Best Practices identifizieren, die sich dieser Kategorie zuordnen lassen:

- Ein RESTful Web-Service sollte nach [Mul12] mindestens zwei verschiedene Repräsentationsformate unterstützen, zum Beispiel JSON oder XML.
- Als Standard ist gemäß [Mul12] JSON als Repräsentationsformat auf Grund steigender Verbreitung zu wählen.
- Bei der Wahl der Repräsentationsformate sollte auf existierende Datenformate aufgesetzt werden, die bereits eine Unterstützung von Hypermedia bieten und demnach für RESTful Web-Services geeignet sind. An dieser Stelle wären drei populäre JSON-basierte Hypermedia-Formate zu erwähnen: JSON-LD, Collection+JSON und Siren.
- Für die Auswahl des MIME-Typen sollte HTTP zur Inhaltsvereinbarung (*Content Negotiation*) genutzt werden, mit dessen Hilfe der Dienstnehmer dem Dienstgeber über das HTTP-Header-Feld `ACCEPT` seine akzeptierten Formate mitteilt. Zudem existiert die Möglichkeit, diese Formate entsprechend der Präferenz des Dienstnehmers mithilfe eines Qualitätsparameters zu gewichten (vgl. [RFC2616], [Vin08]).

Fazit

In diesem Artikel haben wir Best Practices, die in der Literatur verbreitet sind, in Kategorien zusammengefasst und mit Qualitätsmerkmalen der ISO/IEC 25010:2011 (vgl. [ISO]) verknüpft. Durch die steigende Verwendung von RESTful Web-Services in Projekten hat die Gestaltung der Web-Services zunehmenden Einfluss auf die Qualität der gesamten Architektur. Aus diesem Grund müssen auch Web-Services stärker denn je mit besonderer Sorgfalt entwickelt werden. Mit unserer Übersicht geben wir Architekten und Entwicklern ein Werkzeug an die Hand, um RESTful Web-Services systema-

Literatur & Links

[Fie00] R.T. Fielding, Architectural styles and the design of network-based software architectures, University of California, Irvine, 2000

[Fie08] R.T. Fielding, REST APIs must be hypertext-driven, 2008, siehe:
`http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`

[Fie13] R.T. Fielding, Scrambled Eggs, Evolve'13 – The Adobe CQ Community Technical Conference 2013

[Geb11] M. Gebhart, Qualitätsorientierter Entwurf von Anwendungsdiensten, KIT Scientific Publishing 2011

[IANA14] IANA, Media Types, 2014, siehe:
`http://www.iana.org/assignments/media-types/media-types.xhtml`

[Inf11] InfoQ, How REST replaced SOAP on the Web: What it means to you, 2011, siehe:
`http://www.infoq.com/articles/rest-soap`

[ISO] ISO und IEC, ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models

[Jau14] S. Jauker, 10 Best Practices for better RESTful API, 2014, siehe:
`http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/`

[Mul12] B. Mulloy, Web API Design – Crafting Interfaces that Developers Love, 2012, siehe:
`http://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf`

[Pap14] P. Papapetrou, Rest API Best(?) Practices Reloaded, 2014, siehe:
`http://java.dzone.com/articles/rest-api-best-practices`

[Ric13] L. Richardson, M. Amundsen, S. Ruby, RESTful Web APIs, O'Reilly & Associates 2013

[RFC2616] IETF, RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, siehe:
`http://www.ietf.org/rfc/rfc2616.txt`

[RFC6585] IETF, RFC 6585: Additional HTTP Status Codes, siehe:
`http://tools.ietf.org/rfc/rfc6585.txt`

[Vin08] S. Vinoski, RESTful Webservices Development Checklist, in: Internet Computing IEEE Vol. 12 Nr. 6, S. 94-96, 2008

[Web10] J. Webber, S. Parastatidis, I. Robinson, REST in Practice: Hypermedia and Systems Architecture, O'Reilly Media 2010

tisch und effizient auf die Einhaltung von Best Practices untersuchen zu können.
Im zweiten Teil dieses Artikels werden wir einen Bezug zur ISO/IEC 25010:2011 herstellen, um eine Priorisierung der Best Practices zu ermöglichen. Um unsere Ergebnisse direkt in Projekten anwenden zu können, werden wir diese in Form eines Bewertungsbogens zusammenfassen. Dieser wird zum Download bereitstehen und kann pro Web-Service ausgefüllt werden. Unser Ziel ist es, hierdurch die Qualität von Web-Services in Projekten und somit die Qualität der gesamten Architektur systematisch sicherzustellen. ||

Die Autoren

|| Prof. Dr. Sebastian Abeck
(abeck@kit.edu)
leitet an der Fakultät für Informatik des Karlsruher Instituts für Technologie (KIT) die Forschungsgruppe Cooperation & Management (C&M). Seine Forschungsinteressen betreffen SOAs, das Internet der Dinge sowie Identitäts- und Zugriffsmanagement.

|| Dr. Michael Gebhart
(michael.gebhart@iteratec.de)
ist IT-Management-Berater bei der iteratec GmbH. Er unterstützt Kunden bei der Konzeption und Durchführung von IT-Projekten sowie der strategischen Beratung im EAM-Umfeld. Seine Forschungsschwerpunkte sind die Qualitätsanalyse von Web-Services und SOAs.

|| Pascal Giessler
(pascal.giessler@student.kit.edu)
ist Masterstudent am KIT in der Forschungsgruppe Cooperation & Management. Zuvor begleitete er ein Forschungsprojekt für modellgetriebene Softwareentwicklung in der Forschungsgruppe Software Design & Quality (SDQ).