Sonderdruck aus
JavaSPEKTRUM 1/2018

Architekturstil für verteilte Systeme

Warum REST viel mehr ist als ORM über HTTP

Richard Gross

RESTful APIs sind kaum noch aus der Anwendungslandschaft wegzudenken. Ursprünglich eine leichtgewichtige Alternative zum XML-basierten SOAP, sind sie mittlerweile Standard für Programmierschnittstellen. 2010 waren 74 Prozent der öffentlichen APIs RESTful [ROD]. Roy Fielding, der Erfinder von REST, bemerkte jedoch schon 2008, dass die meisten APIs nicht den definierten Kriterien entsprechen [Fie08].

Die technische Qualität der Programmierschnittstelle kann dabei exzellent sein. Viele Entwickler kennen essenzielle Eigenschaften von REST nicht. So verschenken sie Vorteile bezüglich Skalierbarkeit und Entkopplung von Server und Consumer.

Ein RESTful API kostet allerdings auch mehr Aufwand auf Serverseite. Je mehr (unterschiedliche) Consumer eine Programmierschnittstelle hat, desto mehr lohnt sich ein RESTful API. Diesen Trade-off können viele Entwickler jedoch gar nicht eingehen, da sie die Möglichkeiten nicht kennen.

Architekturstil für verteilte Systeme

Der Begriff REST ist Opfer seines eigenen Erfolgs geworden, die semantische Diffusion [Fow06] macht ihn schwammig. Viele Interpretationen von RESTful könnte man als „ORM (Object-Relational Mapping) über HTTP“ bezeichnen: Datenbankobjekte werden per Web-API als JSON zugänglich gemacht. Diese APIs werden im Folgenden als OAR (Object as a Resource) [MoN16] bezeichnet. Dabei ist REST viel mehr, und nicht nur auf APIs anwendbar: Es ist *ein Architekturstil für verteilte Systeme*.

Dieser Artikel beschreibt die Entstehung von REST und die Kerneigenschaften des Stils, wie sie Roy Fielding in seiner Dissertation „Architectural Styles and the Design of Network-based Software Architectures” [Fie00] formuliert. Anschließend thematisiert der Artikel die Auswirkungen des Stils auf eine Programmierschnittstelle und gibt Empfehlungen, wann ein RESTful API sinnvoll ist.

WWW

Das World Wide Web (WWW) ist das wohl bekannteste und erfolgreichste *verteilte System*. Es ist darüber hinaus ein sehr gutes Beispiel für ein iterativ erweitertes System mit sehr vielen Stakeholdern.

Die Kernkomponenten HTTP (Protokoll), HTML (Repräsentation) sowie URI (Ressource) werden seit 1989 erweitert und sind aktuell in den abwärtskompatiblen Versionen HTTP/2, HTML 5 sowie URI 2014 verfügbar. Die Stakeholder haben allerdings sehr widersprüchliche Ideen, was in neue Versionen gehört. Wie kann man nun bewerten, *welche Anforderungen dem Web zuträglich sind* und welche nicht?

Richard Gross arbeitet seit 2012 als IT-Architekt im Bereich IT-Sanierung bei MaibornWolff. Seine Schwerpunkte sind IT-Architekturen und agile Entwicklung. E-Mail: richard.gross@maibornwolff.de

Mit dieser Frage hat sich Roy Fielding intensiv während der Spezifikation von HTTP/1.0 (1996) und HTTP/1.1 (1999) beschäftigt. In der Spezifikationsphase wurde die formal nie festgeschriebene Architektur des Webs *reverse-engineered* und als Grundlage genutzt, um Webkonzepte zu diskutieren. Diesem Architekturstil gab Fielding 2000 in seiner Dissertation [Fie00] den Namen *REST*, kurz für *Representational State Transfer*: ein Stil für verteilte Hypermedia-Systeme wie das WWW. Fielding grenzt REST von anderen Netzwerk-basierten Architekturstilen wie *„Pipes and Filter"*, *„Client-Server"* und *„distributed objects"* ab.

Hypermedia ist eine der Kerneigenschaften von REST, die in den meisten „RESTful" APIs ignoriert wird. Hypermedia bedeutet, dass Daten angereichert sind um Anwendungssteuerungsinformationen, zum Beispiel Metadaten wie Hyperlinks. Der Anbieter beschreibt für den Konsumenten einer Programmierschnittstelle, wie er via Schnittstelle mit der Anwendung interagiert.

Der H-Faktor [CL] eines *Media Type* (SVG, Atom, HTML usw.) gibt dabei an, wie ausgeprägt die Hypermedia-Unterstützung ist. Bei neun möglichen Faktoren gibt es vier Link- und fünf Steuerungsfaktoren. SVG bietet beispielsweise keine Steuerungs- und nur zwei Linkfaktoren: eingehende und ausgehende Links. HTML kann deutlich mehr und deklariert mit Steuerungsfaktoren auch die semantische Bedeutung einer Aktion (GET, POST für Formular). Welchen Media Type man wählt, hat daher eine starke Auswirkung auf die Ausdrucksstärke des Systems. Es kann auch sinnvoll sein, je nach Eigenschaft der Ressource unterschiedliche Media Types zu wählen.

Representational State Transfer

REST ist ein idealisiertes Modell der Interaktionen im Web: ein Netzwerk von Ressourcen, durch die der Benutzer mit einer uniformen Schnittstelle navigieren kann. Das WWW ist ein Beispiel, denn *Http ist eine Implementierung des REST-Stils*.

Theoretisch sind ganz andere Implementierungen denkbar. Roy Fielding arbeitete 2003 sogar an einem binären Protokoll namens Waka [waka] als mögliche HTTP-Alternative. Ein Grund war vermutlich, dass HTTP nicht vollständig REST-konform ist. Die HTTP/1.0-Spezifikation wurde weit vor der Definition von REST fertiggestellt, und HTTP/1.1 sollte abwärtskompatibel bleiben.

Cookies sind beispielsweise nicht REST-konform, da sie es erlauben, vergangene Zustände zu ändern: Der HTTP-Konsument schickt mit seinem GET-Request nicht nur die vorgesehenen Parameter an den Server, sondern auch den Inhalt des Cookies, beispielsweise `"Interesse=Computer"`. Der Server liest die Parameter des GET und des Cookies und antwortet entsprechend. Durch die Anfragen kann der Inhalt des Cookies geändert werden: jetzt `"Interesse=Autos"`. Mit Klick auf den Zurück-Knopf wird das erste GET mit den ursprünglichen Parametern erneut abgeschickt, aber der Konsument übermittelt ein anderes Interesse im Cookie. Damit ändert sich die Antwort des Servers. Wir sind mit dem Cookie durch die Zeit gereist und haben die Vergangenheit geändert.

JavaScript ist dagegen REST-konform, da REST die Möglichkeit von Code-on-demand explizit einräumt.

Roy Fielding behandelt diese Aspekte von REST in seiner 150-seitigen Dissertation ausführlich. Hier geht es nur um die von ihm beschriebenen Ziele.

Ziele von REST

REST zielt auf Skalierung der Komponenteninteraktionen, Generalisierung von Schnittstellen, unabhängig veröffentlichbare Komponenten, gewährleistete Sicherheit und gekapselte Altsysteme.

Um diese Ziele zu erfüllen, hat REST fünf Einschränkungen, hier im Original:

- Resource is unit of identification.
- Resource-generic interaction semantics.
- Resource is manipulated through exchange of representations.
- Self-descriptive messaging.
- Hypermedia as the engine of application state (HATEOAS).

Identifikation der Ressourcen

Die *Kernabstraktion* von REST ist die Ressource. Jede Information, die mit einem Namen versehen werden kann, kann zu einer Ressource werden: ein Dokument, ein Bild, ein temporärer Service wie das heutige Wetter in Frankfurt. Ressourcen werden über eine URI (Uniform Resource Identifier, [Wiki-b]) eindeutig identifiziert. Man kann auf jede Ressource *uniform*, also immer gleich zugreifen, egal auf welchem Server sie liegt. Das ist ein großer Unterschied zu anderen verteilten Systemen mit eigenen Adressierungsmöglichkeiten pro System. Das ist ein großer Unterschied zu anderen verteilten Systemen, in denen jedes System seine eigene Adressierungsmöglichkeit anbietet.

Generische Interaktion

Es gibt eine generische Schnittstelle mit fester Semantik, um auf Daten einer Ressource zuzugreifen und diese zu manipulieren. In der REST-Implementierung HTTP gibt es beispielsweise die spezifizierten Anfragemethoden GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT und PATCH; sie werden HTTP-Verben genannt. Diese Einschränkung von REST ist ein großer Unterschied zu *Remote Procedure Calls* (RPCs) mit spezifischer Schnittstelle pro Information.

Das Flickr-API ist demnach nicht konform mit REST, da es HTTP nur als Tunnelprotokoll nutzt, um spezifische *Procedures* aufzurufen [flickr]. Die einheitliche Semantik geht verloren. Ohne Dokumentation weiß man nicht, welche Methode Daten verändert und welche *safe* ist und nur lesend zugreift. Bei korrekter Semantik kann man Anfragen *cachen* und die Last auf Anwendungen verringern.

Manipulation von Ressourcen durch Austausch von Repräsentationen

REST-Komponenten kommunizieren, indem sie Repräsentationen von Ressourcen austauschen. Diese Repräsentationen in einem standardisierten Format wie HTML oder CSV abstrahieren von der tatsächlichen Ressource und müssen nichts mit dem eigentlichen Aussehen auf Serverseite zu tun haben. Das ist eine Anwendung des Entwurfsmusters *„Separation of Concerns"* und ein ausgeprägter Unterschied zu anderen Architekturstilen, etwa „distributed objects" ([Fie00], Kapitel 3.6.3).

Selbst-beschreibende Nachrichten

Selbst-beschreibende Nachrichten enthalten Information über den Datentyp der Nachricht und wie sie verarbeitet werden muss. In

HTTP ist das beispielsweise gelöst über den Header *„Content-Type: application/xml“*, *„image/jpeg“* oder *„text/html“*. Typischerweise ist der Typ bei der Internet Assigned Numbers Authority registriert [IANA-b]. Das ist jedoch keine Pflicht, man kann auch einen eigenen Typ definieren, nutzen und später bei IANA als *Vendor*-Typ einreichen.

Auch der Content selbst kann eine Detaillierung enthalten. Ein “*text/html*”-Dokument beschreibt sich zusätzlich mit `<!DOCTYPE html>`, damit alle Browser die relevanten Spezifikationen [W3C-c] befolgen. Wären die Nachrichten nicht selbst-beschreibend, müsste man die Informationen aus einer vorab ausgetauschten Spezifikation ziehen.

Hypermedia as the Engine of Application State (HATEOAS)

Wikipedia [Wiki-a] definiert: „In information technology and computer science, a program is described as stateful if it is designed to remember preceding events or user interactions; the remembered information is called the state of the system.”

In einer REST-Architektur merkt sich der Consumer diesen Zustand. Der Server ist dagegen immer *stateless* wie oben definiert. Hypermedia liefert alle möglichen Zustandstransitionen (Ereignisse) für eine Repräsentation. Der Consumer führt eine der Transitionen aus und ändert damit seinen Zustand. REST erlaubt so, einen *Zustandsautomaten* per Schnittstelle nach außen zu tragen. Das ist der vielleicht größte Unterschied zu anderen Architekturen, bei denen der Consumer vorab wissen muss, in welcher Reihenfolge Ereignisse passieren.

Bei REST bleibt die Logik, welche Zustands*übergänge* möglich sind, auf dem Server. Die Information, welche Zustände *durchlaufen* wurden, bleibt auf dem Consumer; nur er ist *stateful*. Im Web-Browser kann man sich den Zustand in der *History* anschauen.

Der Consumer kann die Historie der Zustände nutzen und seine Anfragen anpassen. Der Server wird auf jede Anfrage mit denselben Parametern gleich reagieren. Er ist *stateless* und passt seine Antwort nicht an vorherige Aktionen des Consumer an.

Daraus entsteht eine sehr *lose Kopplung* zwischen Anwendungen, die unabhängig voneinander veröffentlicht und über einen Link verbunden werden können. Das erlaubt eine gute *Skalierung*. Man kann prinzipiell mit jedem Link die Anwendung wechseln und die Last verteilen.

RESTful APIs

RESTful APIs beachten die Einschränkungen des REST-Architekturstils. OAR-APIs erfüllen nur einige Kriterien und ignorieren Hypermedia. Das zeigt sich bereits im Design: Bei RESTful APIs startet man nicht mit Ressourcen, sondern mit Zuständen und Zustandsübergängen [Amu13]. Dieser Zustandsautomat erfüllt alle relevanten und bekannten Anwendungsfälle, die ein Konsument an die Programmierschnittstelle hat. Falls möglich schärft man das Design frühzeitig mit ihm. Bei OAR-APIs ist der Consumer typischerweise auf sich allein gestellt und kann keine Unterstützung erwarten.

Mehr als OAR

Das Richardson Maturity Model [Fow10] klassifiziert Web-APIs in vier Level. Die Stufen bauen aufeinander auf, Level 3 hat beispielsweise alle Eigenschaften von Level 2 und Level 1. Nur die höchste Ausbaustufe ist konform mit den REST-Prinzipien [Fie08]:

- *Level 0:* HTTP als Tunnelprotokoll für RPC (z. B. SOAP oder JSON-Pure APIs [BigLie]).
- *Level 1:* Man kann individuelle Ressourcen adressieren und mit RPCs ansteuern.
- *Level 2* : Man nutzt semantisch sinnvolle Verben (OAR).
- *Level 3:* Hypermedia Controls (REST).

Je nach Design der Programmierschnittstelle kann Level 2 natürlich auch RPC (und damit nicht mehr OAR) sein, zum Beispiel, wenn man den Namen der Methode in die URI packt:

```
GET http://localhost/todo/alleMeineTodos
GET http://localhost/todo/nichtFertigeTodos
POST http://localhost/todo/nichtFertigeTodos/1234
```

Beispiel

Ein RESTful API beginnt immer mit einer bekannten URI, auf die ein GET ausgeführt wird, beispielsweise:

```
GET http://localhost/
```

Man erhält folgende Antwort:

```
200 OK
Expires: Thu, 12Jun2008 17:20:33
Content-Type: application/xhtml+xml
Content-Length: ...

<div class="root">
  <a href="/todo" type="application/hal+xml" rel="todo"/>
  <a href="/favoriten" type="application/hal+xml" rel="favoriten"/>
</div>
```

Die Antwort beschreibt sich über den Content Type selbst. Das Beispiel-API nutzt für die Einstiegsseite den bei IANA registrierten Media Type [IANA-b] XHTML. *Todo* und *Favoriten* setzen dagegen auf *Hal+XML*, um Ressourcen besser zu repräsentieren.

JSON bietet keine Unterstützung für Hyperlinks [W3C-a]. Formate, die JSON um Hyperlinks erweitern, sind beispielsweise Siren [GitH-a], Collections+Json [GitH-b], Hal+Json [HAL] oder Json-Ld [W3C-b].

Durch das *parsen* der XHTML-Antwort entdeckt man Links, um die Programmierschnittstelle weiter zu erkunden. Die Links beschreiben nicht nur das Ziel (`href` - hypertext reference) und den Media Type am Ziel (content `type`), sondern auch die Relation zwischen aktuellem Zustand und Zielzustand (`rel` - relationship). Ein Consumer kann daher die Links durchsuchen und dem ihm bekannten `rel="todo"`-Link folgen:

```
GET http://localhost/todo

200 OK
Expires: Thu, 12Jun2008 17:20:33
Content-Type: application/hal+xml
Content-Length: ...

<resource rel="self" href="/todo">
  <link rel="next" href="/todo?seite=2" />
  <link rel="find" href="/todo{?id}" />

  <resource rel="todo" href="/todo/1">
    <link rel="favorisieren" href="/favoriten/" />
    <name>Auto kaufen</name>
    <status>nicht fertig</status>
  </resource>
  <resource rel="todo" href="/todo/2">
    <name>Blog schreiben</name>
    <status>fertig</status>
  </resource>

  <seite>1</seite>
  <proSeite>2</proSeite>
  <gesamt>33</gesamt>
</resource>
```

Der Content wird diesmal als *Hal+Xml geparst*. Auch dieser Media Type erlaubt es, Links mit Relationen näher zu beschreiben. `next` ist anders als die vorigen Relationen bei der IANA registriert [IANA-a] und sagt aus, dass dieser Zustand Teil einer Serie ist und es auf `seite=2` weitergeht.

Todo/1 kann man favorisieren, *Todo/2* nicht. Grund und Logik hierfür liegen auf dem Server. Der Consumer muss sie weder kennen noch duplizieren. Man hätte an dieser Stelle auch entscheiden können, den Link zum Favorisieren von */todo/1* nicht einzubetten. Dann würde man ihn erst sehen, wenn ein GET ausgeführt wird:

```
GET http://localhost/todo/1

200 OK
Expires: Thu, 12Jun2008 17:20:33
Content-Type: application/hal+xml
Content-Length: ...

<resource rel="self" href="/todo/1">
  <link rel="favorisieren" href="/favoriten/" />
    <name>Auto kaufen</name>
    <status>nicht fertig</status>
  </resource>
```

Dieser kleine Einblick in ein RESTful API zeigt beispielhaft Vor- und Nachteile. Ein längeres Beispiel findet man online [Web08].

Im Folgenden sind die Vor- und Nachteile zusammengefasst.

Vorteile von RESTful APIs

(1) RESTful APIs sind *entdeckbar* und die *Dokumentation ist eingebaut*. Bei OAR-APIs benötigt man eine vollständige *WSDL* oder Swagger-Spezifikation, die vor dem ersten Zugriff ausgetauscht werden muss. Nur so weiß ein Consumer, welche URI er aufrufen muss, um Daten zu betrachten und zu manipulieren. Bei einem RESTful API ist es sinnvoll, die Spezifikation der nicht-standardisierten Relationen (`rel="favorisieren"` usw.) vorab auszutauschen. Ohne diese kann der Entwickler des Consumers die Programmierschnittstelle allerdings auch entdecken und muss keine Links hardcoden.

(2) RESTful APIs *entkoppeln* den Consumer vom Server. Der Server enthält die vollständige Business-Logik und passt mitgeschickte Links an. Der Consumer dupliziert keine Business-Logik und kennt keine Workflows. Der Server liefert alle erlaubten Links. Der Consumer ist dümmer geworden; gleichzeitig muss er mehr über Protokoll, Media Types und Relationen wissen. Dieser Trade-off greift besonders, wenn mehrere Consumer entwickelt werden. Die Business-Logik wird nur im Server implementiert, statt in jedem Consumer.

(3) RESTful APIs erleichtern die *Evolution* der Schnittstelle. Die Logik auf dem Server kann kontinuierlich erweitert werden. Neue Links, Entitäten oder Attribute können im Consumer im Nachhinein nachgezogen werden. Bis dahin ignoriert der Consumer ihm unbekannte Links. Bei Skalierungsprobleme kann man die URIs in der Schnittstelle anpassen und auf neue Server verweisen, beispielsweise wenn URIs von Bildern in ein *Content Distribution Network* ausgelagert werden; oder wenn aus einem Microservice ein zweiter herausgelöst wird. Der Consumer folgt Links und wird ohne Anpassungen im Code weiter funktionieren.

Nachteile

(1) RESTful APIs benötigen *mehr Designaufwand*. Die Programmierschnittstellen müssen für den Consumer entwickelt werden und dessen Anforderungen berücksichtigen. Kennt man den Consumer nicht oder kaum, ist das schwierig.

(2) Für RESTful APIs gibt es wenig *Tool-Support*, das erhöht den Entwicklungsaufwand. Frameworks wie Spring erlauben es, mit wenigen Zeilen Code Datenbank-Entitäten als JSON-Objekte über eine Webschnittstelle bereitzustellen. Entwickler von RESTful APIs profitieren nicht von dieser maximalen Kopplung der Web-Programmierschnittstelle an die Datenbank. Und: Man kann wenig automatisieren. Es muss viel Code zum Mappen von Business-Objekten auf Ressourcen und Repräsentationen geschrieben werden. Spring Hateoas [Hateoas] und JAX-RS [J7Link] bieten Funktionen, um die Programmierschnittstelle um die essenziellen Hyperlinks zu ergänzen.

(3) RESTful APIs sind *weniger ausdrucksstark* als RPC. RPC-Programmierschnittstellen können sehr expressive Methodennamen benutzen, die die Domänenbegriffe spiegeln. Ein RESTful API setzt dagegen auf eine uniforme Schnittstelle.

(4) RESTful APIs sind *weniger effizient*, sie schicken durch potenziell lange Links größere Repräsentationen übers Netzwerk. Allerdings ist die Größe der Repräsentation vor allem für Firmen mit sehr vielen Anfragen interessant, Google zum Beispiel mit zwei Trillionen Suchanfragen pro Jahr [Sul12], oder für Netflix als Dienst, der in Nordamerika ein Drittel der Bandbreite verbraucht [Hugh16]. Wer diese Skalierungsprobleme hat, kann sich wirklich glücklich schätzen. Alle anderen können durch Aktivieren von gzip Komprimierungsraten von 70 bis 90 Prozent [Gri] erreichen. Die zusätzlichen Metadaten wiegen dann weniger stark.

Bewertung

REST als Architekturstil des WWW beschreibt, wie Anwendungen lose gekoppelt, unabhängig veröffentlicht und skaliert werden können.

Semantische Diffusion

Dieser Artikel bezeichnet – entsprechend dem Richardson Maturity Models – Web-APIs bis Level 2 als OAR (*Object as a Resource*). Nur Level 3 entspricht den REST-Kriterien von Roy Fielding und wird in diesem Artikel als RESTful bezeichnet. Andere sprechen bei Level 3 von Hypermedia-APIs und bezeichnen alles andere weiterhin als „REST“ („REST is dead, say Hypermedia API instead“, [Kla12]). Dadurch herrscht allerdings weiterhin *vollkommene Unklarheit* darüber, was „REST“ für die jeweilige Programmierschnittstelle bedeutet. Ist Level 0, 1 oder 2 gemeint? Die einzige Konstante der schwammig definierten „RESTful“ APIs ist die Verwendung des Media Type JSON. Dabei ist gerade JSON unvereinbar mit *Representational State Transfer*: Es bietet keine Unterstützung für Hypermedia, um den *State Transfer* zu modellieren.

Die Unterschiede zwischen OAR und RESTful

Wenn man den REST-Stil auf eine Schnittstelle anwendet, erhält man eine leicht weiterentwickelbare Programmierschnittstelle mit eingebauter Dokumentation, die den Entwicklungsaufwand für Consumer senken kann. Die Zustandsübergänge sind in der Verantwortlichkeit des Servers und werden nur dort implementiert: DRY (*don't repeat yourself*). Dafür steckt man auf Serverseite Zeit in Design und Entwicklung und hat eine verringerte Effizienz der Nachrichten.

Am Beispiel konnte man sehen, dass RESTful APIs dem Design von Webseiten stark gleichen. Das ist gewollt. Ein RESTful API ist

eine Webseite für Maschinen: Consumer mit limitiertem Vokabular. RESTful APIs haben damit ein starkes Alleinstellungsmerkmal. Sie braucht als einzige Programmierschnittstelle initial nur die Einstiegs-URI des Service, standardisierte Mediatypen und Protokolle. Das grenzt RESTful APIs von OAR-APIs ab, die Schnittstellenspezifikation inklusive aller URIs und Workflows vorab austauschen müssen.

Der große Unterschied zwischen OAR und RESTful fällt im Designprozess auf. Bei OAR fokussiert man auf Ressourcen und bringt alle benötigten Daten zum Consumer, der seine Anwendungsfälle lokal erfüllt. Bei RESTful wird ein Zustandsautomat publiziert. Man arbeitet mit dem Consumer und löst den Anwendungsfall auf dem Server.

Fazit

Ein REST-Ansatz bietet Vorteile, wenn die Programmierschnittstelle von bekannten Consumern genutzt wird, eine lose Kopplung der Consumer gewünscht ist oder alle Möglichkeiten der Skalierung offen bleiben sollen.

Wenn viele Consumer mit unbekannten Anwendungsszenarien auf das API zugreifen oder der Server nur eine reine Datensenke mit wenig Business-Logik ist, lohnt sich eher der OAR-Ansatz.

Weiterführende Infos bieten [Fie00], [YouT], [Gie16], [Cha16], [Sou15], [WPR10], [RiAm13] und [Amu11].

Literatur und Links

[Amu11] M. Amundsen, Building Hypermedia APIs with HTML5 and Node, O'Reilly, 2011

[Amu13] M. Amundsen, Designing and Implementing Hypermedia APIs, InfoQ, 5.2.1013, https://www.infoq.com/articles/hypermedia-api-tutorial-part-one

[BigLie] M. S. Mikowski, RESTful APIs, the big lie, https://mmikowski.github.io/the_lie/

[Cha16] Blog von J. Channon, What is a Hypermedia client, 28.4.2016, http://blog.jonathanchannon.com/2016/04/28/what-is-a-hypermedia-client/

[CL] Control Data for Links, http://amundsen.com/hypermedia/hfactor/#cl

[Fie00] R. Th. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation, University of California, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/abstract.htm

[Fie08] R. T. Fielding, REST APIs must be hypertext-driven, 20.10.2008, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[flickr] https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos

[Fow06] M. Fowler, SemanticDiffusion, 14.12.2006, https://martinfowler.com/bliki/SemanticDiffusion.html

[Fow10] M. Fowler, Richardsons Maturity Model, 18.3.2010, https://martinfowler.com/articles/richardsonMaturityModel.html

[Gie16] Blog von O. Gierke, The Benefits of Hypermedia APIs, 29.4.2016, http://olivergierke.de/2016/04/benefits-of-hypermedia/

[GitH-a] https://github.com/kevinswiber/siren

[GitH-b] https://github.com/collection-json/spec

[Gri] I. Grigorik, Codierung und Übertragungsgröße textbasierter Ressourcen optimiert, https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer?hl=de

[HAL] Hypertext Application Language, http://stateless.co/hal_specification.html

[Hateoas] Spring HATEOAS, http://projects.spring.io/spring-hateoas/

[Hugh16] N. Hughes, Netflix boasts 37 % share of Internet traffic in North America, compared with 3 % for Apple's iTunes, 20.1.2016, http://appleinsider.com/articles/16/01/20/netflix-boasts-37-share-of-internet-traffic-in-north-america-compared-with-3-for-apples-itunes

[IANA-a] Link Relations, Internet Assigned Numbers Authority (IANA), https://www.iana.org/assignments/link-relations/link-relations.xhtml

[IANA-b] Media Types, Internet Assigned Numbers Authority (IANA), https://www.iana.org/assignments/media-types/media-types.xhtml

[J7Link] Class Link, Java EE 7, http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Link.html

[Kla12] Blog von St. Klabnik, REST is OVER!, 23.2.2012, http://blog.steveklabnik.com/posts/2012-02-23-rest-is-over

[MoN16] A. Moore-Niemi, OAR is not REST, 8.11.2016, http://mooreniemi.github.io/rest/apis/2016/11/08/oar-is-not-rest.html

[RiAm13] L. Richardson, M. Amundsen, RESTful Web APIs, O'Reilly, 2013

[ROD] Resource Oriented Design, Cloud APIs, Google Cloud Platform, https://cloud.google.com/apis/design/resources

[Sou15] Folien von Paulo Gandra des Sousa: Benefits of Hypermedia APIs, 25.11.2015, https://www.slideshare.net/pagsousa/benefits-of-hypermedia-api

[Sul12] D. Sullivan, Google now handles at least 2 trillion searches per year, 24.5.2012, http://searchengineland.com/http://searchengineland.com/google-now-handles-2-999-trillion-searches-per-year-250247

[W3C-a] Basic Concepts, JSON, World Wide Web Consortium (W3C), https://www.w3.org/TR/json-ld/#basic-concepts

[W3C-b] JSON-LD 1.0, World Wide Web Consortium (W3C), https://www.w3.org/TR/json-ld/

[W3C-c] The DOCTYPE, HTML5, World Wide Web Consortium (W3C), https://www.w3.org/TR/html5/syntax.html#the-doctype

[waka] http://gbiv.com/protocols/waka/

[Web08] J. Weber u. a., How to GET a Cup of Coffee, InfoQ, 2.10.2008, https://www.infoq.com/articles/webber-rest-workflow

[Wiki-a] State, https://en.wikipedia.org/wiki/State_(computer_science)

[Wiki-b] URI, http://de.wikipedia.org/wiki/Uniform_Resource_Identifier

[WPR10] J. Webber, S. Parastatidis, I. Robinson, REST in Practice: Hypermedia and Systems Architecture, O'Reilly, 2010

[YouT] Musik-Video: You Give REST a Bad Name, https://www.youtube.com/watch?v=nSKp2StlS6s