# RESTful APIs - Eine Übersicht

Michael Dazer
michael.dazer@googlemail.com
Technical University Berlin
Berlin Germany

***Zusammenfassung*—Das *Web* zeichnet sich durch ein enormes Wachstum aus. Ständig entstehen neue innovative, verteilte Systeme. Allerdings ist die Web-Service-Landschaft alles andere als übersichtlich. Zum größten Teil findet man spezielle Lösungen, die schwer erweiter- und wiederverwendbar sind. Durch applikationsspezifische Schnittstellen wird bei der Erstellung von *Mashups*[1] Spezialwissen und massiver Zeitaufwand benötigt. Seit einigen Jahren zeichnet sich ein Trend hin zu *RESTful* Web Services ab. *REpresentational State Transfer* (REST) verspricht Wiederverwendbarkeit, Erweiterbarkeit, Skalierbarkeit und die einfache Integration in die vorhandene Netz-Struktur. Dieser Artikel soll als kurzer Überblick über das Thema REST dienen.**

## Einführung

Dieser Artikel soll einen Überblick über das Thema REST bzw. *RESTful* Application Programmable Interfaces (APIs) verschaffen. Mit dem Abschnitt *Stand der Technik* wird eine kleine Übersicht über die gängigsten Methoden, Protokolle und Architekturen im Bereich der *Web Service* Entwicklung gegeben. Service Oriented Architecture (SOA) und Ressource Oriented Architecture (ROA) werden vorgestellt und charakterisiert. Des Weiteren gibt eine Statistik Aufschluss über die aktuelle Verteilung der Architekturen am Markt. Sie macht auch auf die derzeitigen Schwachstellen in diesem Bereich aufmerksam. Im Abschnitt *Motivation* werden diese und weitere Schwachstellen noch einmal zusammengefasst. REST versucht die meisten dieser Probleme durch bestimmte Vorgaben zu vermeiden. Im Abschnitt *Representational State Transfer* wird REST genau beschrieben und darauf eingegangen, worauf beim Entwickeln von *RESTful Web Services* und speziell deren APIs zu achten ist. Es werden die Bedingungen aufgeführt, die ein *RESTful Design* erfüllen muss. Schlagwörter wie Ressourcen, Repräsentationen und *Hypermedia* werden erklärt. Außerdem wird auf typische Missverständnisse der REST-Bedingungen eingegangen, die immer wieder aufkommen. Anschließend wird ein prominentes Beispiel eines *RESTful Web Services* vorgestellt und in Hinsicht der REST Bedingungen untersucht. Zum Abschluss wird das gesammelte Wissen nochmals zusammen gefasst und reflektiert.

## Stand der Technik

In den letzten Jahren haben *Web Services* eine enorme Bedeutung erlangt. *Smartphones* und *Tablets* treiben das Thema weiter voran. Bestellungen über *Amazon*, Datenablage per *Dropbox* oder die komplette Steuerung des Hauses mittels *iPhone App*. Moderne *Web Services* machen es möglich - völlig unabhängig vom Standort oder der verwendeten *Hardware*.
Konzeptionell lassen sich *Web Services* meist in zwei Architekturen einordnen. Bei SOA liegt, unter Verwendung von Protokollen wie Simple Object Access Protocol (SOAP) oder Extensible Markup Language-Remote Procedure Call (XML-RPC), der Fokus bei der Entwicklung auf den Funktionen. Da diese Architektur schon relativ lange existiert, haben sich in diesem Bereich Tools, Prozesse und Methodiken, sogenanntes *Standardwork* etabliert. Allerdings entstehen so oft große und unhandliche Systeme, die zwar hoch optimiert werden können, allerdings relativ schwer erweiterbar sind. SOAP, Common Object Request Broker Architecture (CORBA), Universal Description, Discovery and Integration (UDDI), Distributed Component Object Model (DCOM) und Remote Method Invocation (RMI) sind Technologien aus dem Bereich SOA. Methoden-Aufrufe erfolgen dabei über XML-basierte Nachrichten, die an einen *Service-Endpunkt* gesendet werden. Dort wird die Nachricht *deserialisiert* und an die jeweilige Funktionalität weitergeleitet.
REST ist in den Bereich ROA einzuordnen, wo Daten bzw. Ressourcen im Vordergrund stehen. Im Vergleich zu SOA gibt es keine Frameworks oder Beschreibungssprachen wie Web Services Description Language (WSDL), die bei der Entwicklung unterstützen und leiten. Allerdings muss nicht zwangsläufig für jede kleinere Applikation ein kompletter *Web Service Protocol Stack* verwendet werden. Durch die Ressourcenorientierung ist es möglich, einen Satz an Basisoperationen festzulegen, was wiederum die Übersichtlichkeit und Erweiterbarkeit der Schnittstelle mit sich bringt. Im Gegensatz zu SOA werden Nachrichten nicht an einen festen *Service-Endpunkt* gesendet, sondern es wird mit einigen wenigen Funktionen direkt auf adressierbaren Ressourcen operiert.
Eines haben SOA und ROA gemeinsam, sie entkoppeln Client und Server voneinander. Einen sehr guten Vergleich zwischen den beiden Technologien bietet der Artikel „RESTful Web Services vs. „Big“Web Services“ [1] von Pautasso, Zimmermann und Leymann. Während noch vor einigen Jahren die *großen Web Services* auf Basis von SOAP dominierten, ist momentan ein deutlicher Trend hin zu *RESTful Web Services* zu erkennen.

| | SOA | ROA |
|---|---|---|
| Flexibilität | ✓ | ✓ |
| Für kleinere Projekte gut geeignet | - | ✓ |
| Für große Projekte gut geeignet | ✓ | - |
| Standardwork | ✓ | - |
| Einfache Integration in Web Intermediaries | - | ✓ |

Tabelle I
SOA vs. ROA

Ein beachtlicher Anteil bezeichnet sich allerdings als *RESTful*, obwohl bei näherer Betrachtung Verletzungen essenzieller Eigenschaften sichtbar werden. Diese Missverständnisse sind einerseits Folge des Technologiewandels, andererseits eines Mangels an *Standardwork* zum Thema *RESTful Design*.„In dem Artikel RESTful Web Services Development Checklist von“ [2] wird zwar auf JAX-RS, eine Java API für die Entwicklung *RESTful Web Services* hingewiesen. Solche Ansätze sind allerdings noch selten und vor allem unpopulär. Missverständnisse im Bereich *RESTful-Design* spiegelt auch eine Statistik von Maria Maleshkova, Carlos Pedrinaci und John Domingue [3]

[1]Mashups sind Erweiterungen oder Kompositionen vorhandener Web Services.

aus dem Jahre 2010 wider. In dieser wurden 222 Web APIs, die unter *Programmable-Web*[2] gelistet waren, analysiert und bezüglich Programmier-Paradigmen, Dokumentation, Methoden-Aufrufen und Datenformaten verglichen. Rund fünfzig Prozent der Web Services war im *RPC-Stil* implementiert, die andere Hälfte teilten sich *RESTful* (30%) und hybride Versionen (20%). Es wurde bemängelt, dass bei der Nutzung und Weiterverwendung von derzeitigen Web API, bedingt durch die äußerst unterschiedliche Web-Service-Landschaft, ein enormer zeitlicher Aufwand betrieben werden muss. Unterschiede gab es hauptsächlich bei Methoden-Aufrufen, Dokumentation und Authentifizierung. Hinzu kamen generelle Probleme wie Unterspezifikation und veraltete Beschreibungen. Meist waren zur Entwicklung eigener Applikationen oder *Mashups* spezielle Entwickler-Konten bei den jeweiligen Service-Betreibern notwendig. Primär forderten die Autoren der Statistik ein klareres Bild des Entwicklungsprozesses von Web Services, eine Vereinheitlichung der Dokumentation und eine Steigerung der Wiederverwendbarkeit durch standardisierte Methoden-Aufrufe. Gerade standardisierte Zugriffsmethoden sind wichtig, da der größte Teil der Web Services methoden- und nicht ressourcenorientiert geschrieben ist und somit während der Entwicklung von Applikationen oder *Mashups* eine Menge Zeit damit verbracht wird, eine passende Anbindung zu schaffen.
Der Titel dieses Artikels lautet *RESTful* APIs und nicht *RESTful Web Services*. Hier sei nur darauf hingewiesen, dass die Bezeichnungen *Web Service*, verteilte Anwendung und API eigentlich verschiedene Begriffe darstellen, in diesem Artikel in Bezug auf REST allerdings synonym verwendet werden. Ein *RESTful* entworfener *Web Service* setzt eine *RESTful API* voraus und umgekehrt.

## MOTIVATION

*„First generation web services are like first generation Internet connections. They are not integrated with each other and are not designed so that third parties can easily integrate them in a uniform way.“* [4] - Paul Prescod
Roger L. Costello schreibt in „Building Web Services the REST Way“ [5]:
*„The motivation for REST was to capture the characteristics of the Web which made the Web successful. Subsequently these characteristics are being used to guide the evolution of the Web.“*
Wie in dem Abschnitt *Stand der Technik* angesprochen, können Web Services auf Basis von SOA relativ gut auf ein abgeschlossenes System zugeschnitten und optimiert werden. Es gibt *Tools*, klare Methoden und Prozesse. Nicht nur das *Software-System* selbst obliegt der Kontrolle des Architekten. Auch *Web Intermediaries*, wie *Firewalls, Proxies* und *Caches* können exakt angepasst werden. Abgeschlossene und optimierte Systeme sind meist für einen bestimmten Zweck vorgesehen und nicht dafür geeignet, einfach erweitert oder wiederverwendet zu werden.
Der Großteil des *Webs* ist allerdings durch Wiederverwendung und Erweiterung gewachsen. REST setzt genau dort an. Schnittstellen sollten übersichtlich gehalten werden, um Erweiterungen nicht zum Opfer zu fallen. Viele *Web Services* sind Erweiterungen oder Kompositionen mehrerer kleiner, unabhängiger *Web Services*, sogenannte *Mashups*. REST erleichtert durch seine kompakten Schnittstellen und Adressierbarkeit die Komposition einzelner Subsysteme. Hält man sich an die Bedingungen die REST stellt, können *Mashups* bzw. *Clients* entstehen, *„die nie eingeplant waren“* [6]. REST unterstützt außerdem die problemlose Integration einer verteilten Anwendung in vorhandene *Web Intermediaries*[3].
SOA bringt trotz seiner Vorteile eine Reihe von Problemen mit sich, u.a. das Problem der Adressierung. Objekte oder Funktionen können nicht direkt adressiert werden, d.h. Referenzen können nicht gespeichert oder weitergegeben werden. Weiterhin müssen sogar Nutzdaten analysiert werden, um *Proxy- oder Firewall-Funktionalität* zu unterstützen. Ein weiterer Nachteil ist die völlige Freiheit, Methoden zu definieren [7]. Dies führt zu absolut unterschiedlichen APIs, was wiederum die Wiederverwendung erschwert.
Die große Anzahl an *REST-RPC-Hybriden* liegt mit hoher Wahrscheinlichkeit an Missverständnissen in der Anwendung von REST, da REST sehr abstrakt und kompakt gehalten ist. Erschwerend sind außerdem mangelnder *Standardwork* und *Tool-Unterstützung*. REST liefert Ansätze, verteilte Anwendungen erweiterbar, wiederverwendbar und skalierbar zu gestalten. Um diese Vorteile nutzen zu können, muss man sich allerdings intensiv mit *Fieldings Architektur* auseinander setzen.

## REPRESENTATIONAL STATE TRANSFER

Der Architektur-Stil REST wurde erstmals im Jahre 2000 von Roy Thomas Fielding in seiner Dissertation mit dem Titel „Architectural Styles and the Design of Network-based Software Architectures“ [8] definiert. REST steht für REpresentational State Transfer. Es handelt sich dabei um einen Architektur-Stil und nicht um eine Methodik oder einen Prozess, da Fielding nur abstrakte Bedingungen vorgibt, jedoch keine expliziten Prozesse, Protokolle oder gar Medien. Es wurden typische Architekturen aus dem Bereich verteilter Anwendungen an Hand gewisser Eigenschaften verglichen. Auf dieser Grundlage schuf Fielding eine Architektur, deren Fokus auf Skalierbarkeit, Erweiterbarkeit und *Interoperabilität* liegt. Als Beispiel für ein *RESTful Design* wird gerne das HTTP-Protokoll herangezogen, an dessen Entwicklung Fielding maßgeblich beteiligt war. Ähnlich wie beim *Browsen* im Netz, bewegt sich der Benutzer mittels *Links* von Inhalt zu Inhalt durch die Applikation. Ein wichtiger Design-Aspekt war die effiziente Nutzung von *Web Intermediaries* und die sinnvolle Einbettung erfolgreicher Technologien wie HTTP, HyperText Markup Language (HTML) oder XML. Folgende Bedingungen liegen dem REST-Stil zu Grunde:

- Client-Server
- Stateless
- Caching
- Uniform Interface
- Layered System
- Code on Demand

Durch Kapselung von *Client* und *Server* können diese getrennt voneinander entwickelt, bzw. der *Client* kann so einfacher portabel gehalten werden. Der *Server* fällt schlanker aus und gewinnt an Skalierbarkeit.
Mit *Stateless* bzw. Zustandslosigkeit ist nicht gemeint, dass *Client* und *Server* ihren Zustand nicht wechseln dürfen. Allerdings muss muss jede Operation in sich abgeschlossen sein. *„REST schreibt vor, dass der Zustand entweder vom Client gehalten oder vom Server in einen Ressourcenstatus umgewandelt wird.“* [9] Dies birgt mehrere Vorteile. Die API wird klar und deutlich gehalten und es muss nicht aus dem Kontext ermittelt werden, was eine Anfrage bedeutet. *Links* auf Ressourcen können ohne weiteres gesichert und wieder

[2] http://www.programmableweb.com

[3] *Web Intermediaries* sind Zwischenstationen wie *Proxies* oder *Gateways* zwischen zwei Endpunkten

verwendet werden. Zustandslosigkeit wirkt sich auch in Verbindung mit *Caching* positiv auf Stabilität und Skalierbarkeit aus, da keine separaten Sitzungsdaten benötigt werden.
Das *Caching* selbst senkt die Latenz und das aufkommende Datenvolumen.
Eine zentrale Rolle nimmt das Schlüsselwort *Uniform Interface* ein. In seiner Dissertation [8] schreibt Fielding:
*„In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state."*
Mit *Hypermedia* sind grundsätzlich multimediale Inhalte des *Webs* gemeint. Es setzt sich aus den beiden Worten *Hypertext* und Multimedia zusammen. Die letzte Bedingung in dem Zitat: *„hypermedia as the engine of application state"* wird in der Literatur mit HATEOAS abgekürzt. Mehr dazu im Abschnitt *URIs und Hypermedia*. REST ist eine ressourcenorientierte Architektur und steht somit im Gegensatz zu den meisten Implementierungen aus dem Bereich SOA. Um sich durch Ressourcen zu navigieren, fordert Fielding ein schlankes, gemeinsames *Interface*. Außerdem verlangt er, dass die Applikation nicht an ein Protokoll gebunden sein darf, da dies zu stark koppelt. Meist wird in der Literatur allerdings auf REST in Verbindung mit HTTP eingegangen. Unter Verwendung von HTTP steht *Uniform Interface* für die Verwendung der HTTP-Methoden GET, POST, PUT und DELETE. Man sollte seine gesamte Applikation mit diesen Basisoperationen entwerfen. Das klingt erst einmal hinderlich, birgt allerdings den Vorteil einer sehr einfachen und übersichtlichen Schnittstelle. Sicherlich ist es mit einigem Aufwand verbunden, seinen gesamten *Service* mittels vier Funktionen zu implementieren. Doch wird durch genau diesen Ansatz das Risiko vermindert, dass die Schnittstelle im Nachhinein angepasst werden muss. Außerdem gab es mit vielen SOA-Implementierungen ein logistisches Problem. SOAP z.B. nutzte ausschließlich HTTP-POST für Funktionen. Ohne eine Anfrage zu untersuchen, war es einem Gateway unmöglich, Zugriffe auf bestimmte Teile der Applikation zu verbieten.
Auf die letzten beiden Punkte, *Layered System* und *Code On Demand* wird in der Literatur relativ wenig eingegangen. Mit *Layered System* wird ein geschichteter, modularer Aufbau gefordert, der der Wartung, Wiederverwendbarkeit und Erweiterbarkeit dient. *Code On Demand* dient einer stärkeren Entkopplung von *Client* und *Server*, der Erweiterbarkeit und Skalierbarkeit, da je nach Bedarf *Code* auch auf den *Client* ausgelagert werden kann.
Der zentrale Aspekt von REST ist die Forderung nach einer gemeinsamen Schnittstelle. Um dieser Forderung nachzukommen, stützt sich REST auf folgende Elemente:

- Eindeutig identifizierbare Ressourcen
- URIs bzw. Verknüpfungen
- Standard HTML Funktionen
- Verschiedene Repräsentationen der Ressourcen

*Ressourcen*

REST ist ressourcenzentriert. Mit dem Abfragen, Erstellen, Bearbeiten und Löschen von Ressourcen wechselt der *Client* seinen Zustand. Ressourcen sind die Grundbausteine einer *RESTful* API. Eine Ressource muss adressierbar sein und kann mehrere Repräsentationen haben.
*„Eine bestehende fachliche Domäne auf einen Schnittstellenentwurf abzubilden, der sich an Ressourcen orientiert, erfordert für jemanden, der in der Vergangenheit klassenorientierte Schnittstellen gewohnt war, ein Umdenken. Kritisch ist dabei, Ressourcen nicht als Konzepte einer Persistenzschicht, sondern als übergreifende, nach außen sichtbare Anwendungskonzepte zu begreifen."* [9] - Stefan Tilkov

*Repräsentationen*

Zu jeder Ressource gehört mindestens eine Repräsentation. Ein *Browser* z.B. wird von einem *Web Service* sicherlich ein HTML Dokument anfordern, um die Ressource vernünftig darstellen zu können, während eine *App* z.B. XML oder JSON verwenden würde. Das eigentliche Auswählen der *Media Types* sollte allerdings nicht per Parameter, sondern mittels *Content Negotiation* erfolgen. Beim *Content Negotiation* wird aus den Header-Informationen der Anfrage ermittelt, um was für einen *Client* es sich handelt und welche Repräsentation dieser verlangen könnte. Alternativ kann im HTTP Header explizit das Attribut „Content-Type"gesetzt werden, mit welchem die Repräsentation direkt gewählt wird. Durch die Unterscheidung zwischen Ressource und Repräsentation können Daten und Darstellung getrennt werden, was eine lose Kopplung erreicht. Es können sehr einfach neue Repräsentationen hinzu gefügt werden, ohne die Logik zu beeinflussen und anders herum.

*XML und JSON:* Laut der Studie „Investigating Web APIs on the WorldWideWeb" [3] von Maria Maleshkova, Carlos Pedrinaci und John Domingue dominieren die beiden Formate XML und JSON unter den Datenaustausch-Formaten im Bereich der *Web Services*. JSON kommt aus der JavaScript-Welt und steht für JavaScript Object Notation. JSON wurde für eine möglichst einfache Übertragung von Datenstrukturen entwickelt, bietet allerdings im Gegensatz zu XML keine Validierung. XML bietet dagegen *Features* wie Namensräume und validierbare Schemata, ist aber deutlich komplexer in der Definition und Anwendung. Durch Namensräume können Attribute und deren Gültigkeit strukturiert werden. Man kann ein vorhandenes XML Schema verwenden oder ein eigenes erstellen. Letzteres birgt allerdings das Problem, dass das Format nicht allen Teilnehmern bekannt ist. In diesem Fall kann man sein Schema bzw. seinen *Media Type* auch bei der IANA[4] registrieren, was relativ aufwändig und zeitintensiv ist. *JSON-Bindings* gibt es in vielen Programmiersprachen. Mit Python würde man die JSON-Daten aus dem XML-Listing 1 wie folgt erzeugen:

```
>>> import simplejson
>>> data = {"name" : "Peter", "options" : [1,0,0,0,1,0,1,0,0]}
>>> simplejson.dumps(data)
'{"name": "Peter", "options": [1, 0, 0, 0, 1, 0, 1, 0, 0]}'
```

Listing 1. JSON Datensatz

XML bietet sich auch dafür an, unstrukturierte Daten zu beschreiben, was mit JSON nicht so leicht möglich ist.

*HTML, CSV, JPG, plain text, etc.:* Auch andere Formate sind möglich. Der Vorteil von HTML ist, dass die API direkt mit dem *Browser* getestet werden kann. Bilder können direkt mittels JPG und Messdaten mit CSV abgerufen werden.

*URIs und Hypermedia*

Ressourcen müssen identifizierbar bzw. adressierbar gemacht werden. Diese Identifikation fordert Fielding mittels URI. URI steht für Uniform Ressource Identifier und ist in RFC 3986 [10] spezifiziert. Im Zusammenhang mit URI fallen auch öfter die Begriffe URL (Uniform Ressource Locator) und URN (Uniform Ressource Name). URI fasst die beiden als Oberbegriff zusammen. Im Gegensatz zu URN müssen URL dereferenzierbar, d.h.

[4]Internet Assigned Numbers Authority; http://www.iana.org/

ohne Nachschlagen in Verzeichnissen erreichbar sein. Für mehr Informationen zu diesem Thema sei auf das Requests For Comments (RFC) verwiesen.
*Hypermedia* steht für multimediale Inhalte in Verbindung mit *Hypertext*. Generell versteht man unter *Hypertext* miteinander verknüpfte und nicht sequenzielle Inhalte, in denen man sich als Nutzer frei bewegen kann. *„Hypertext beschränkt sich nicht nur auf HTML. Anwender oder autonome Systeme können Links auch einfach nur folgen, vorausgesetzt, sie verstehen das Datenformat und die Relationen.“* [11]
URI sind ein wichtiges Element in REST. Über eine URI kann man eine bestimmte Ressource erreichen und dies mittels Zustandslosigkeit ohne Hintergrundinformationen. Ressourcen selbst können und sollen wiederum weitere *Links* enthalten. Wir erinnern uns an HATEOAS. Die Ressourcen treiben den Anwendungszustand. Stefan Tilkow beschreibt dies in REST und HTTP wie folgt: *„Der Client bewegt sich damit durch eine Menge von Seiten; welche dies sein können, wird vom Server vorgegeben und damit begrenzt; welche konkret angefordert werden, entscheidet der Client (bzw. dessen Benutzer). Zu jedem Zeitpunkt haben die Ressourcen des Servers einen definierten Status.“* [9] Wie dies aussehen könnte, zeigt ein kleines Beispiel:

```
POST www.autoteile.de/shop/warenkoerbe <Benutzerdaten>
 ==> HTTP/1.1 201, Created
 ==> www.autoteile.de/shop/warenkoerbe/1234
 ==> www.autoteile.de/shop/teile/
 ==> www.autoteile.de/shop/teile/kategorien/

GET www.autoteile.de/shop/teile/kategorien
 ==> HTTP/1.1 200, OK
 ==> ...
 ==> www.autoteile.de/shop/teile/kategorien/reifen
 ==> www.autoteile.de/shop/teile/kategorien/bremsen
 ==> ...

GET www.autoteile.de/shop/teile/kategorien/bremsen
 ==> HTTP/1.1 200, OK
 ==> ...
 ==> www.autoteile.de/shop/teile/bremsen/4678
 ==> www.autoteile.de/shop/teile/bremsen/4679
 ==> ...

GET www.autoteile.de/shop/teile/bremsen/4678
 ==> HTTP/1.1 202, Accepted
 ==>
   <Artikel>
     <Name>Anker 2.0</Name>
     <Beschreibung>
       Ab jetzt nur noch Vollbremsungen...
     </Beschreibung>
     <Anmerkung>Beläge für je einen Sattel</Anmerkung>
     <Preis>32.80</Preis>
     <Verwandte Artikel>
       <Artikel xlink:href=www.autoteile.de/shop/teile/bremsen/4679>
         <Beschreibung>
           Halten ein Autoleben lang...
         </Beschreibung>
       </Artikel>
     </Verwandte Artikel>
   </Artikel>

PUT www.autoteile.de/shop/warenkoerbe/1234 <4678, 4679>
 ==> HTTP/1.1 202, Accepted

DELETE www.autoteile.de/shop/warenkoerbe/1234/teile/4679
 ==> HTTP/1.1 204, No Content

POST www.autoteile.de/shop/bestellungen 1234
 ==> HTTP/1.1 201, Created
 ==> www.autoteile.de/shop/bestellungen/5678
```

Listing 2. Beispiel Onlineshop

Zuerst wird mittels HTTP-POST und unter Angabe der Benutzerdaten ein Warenkorb erstellt. Zurückgeliefert wird ein *Status-Code*, ein *Link* auf den erstellten Warenkorb und *Links* auf Teile oder Kategorien von Teilen. Das ist der Ansatz von HATEOAS. Durch *Links*, bzw. URIs werden dem *Client* eine Reihe von Zustandsübergängen angeboten. Er wird so intuitiv durch die Applikation geführt. Über die Teile-Kategorien kann bis zu den einzelnen Teilen navigiert werden. Wichtig dabei ist, dass jede Operation einen Status-Code und weitere *Links* liefert, frei nach dem Motto *"Dies könnte Sie interessieren"*.

Fielding weist in „REST APIs must be hypertext-driven“ [11] darauf hin, dass man seine Zeit nicht in den Aufbau hierarchischer und leserlicher URL-Schemata, sondern in die Definition verständlicher und aussagekräftigen Medien Typen verwenden sollte. Des Weiteren sollte nicht mit festen, sondern viel mehr mit logischen URI gearbeitet werden, die der Server selbst verwaltet. Der Entwickler sollte nicht davor zurückschrecken URIs anzupassen. Diese Änderungen können für den *Client* durch *Forwarding* transparent gehalten werden. Ist dies nicht möglich sollte mit HTTP-Status-Codes wie 301 oder 307 (siehe Tabelle III) in Verbindung mit alternativen *Links* gearbeitet werden.

*Funktionen*

Auf die mittels URI adressierbaren Ressourcen wird mit generischen Funktionen zugegriffen. Mit GET können Repräsentationen angefragt werden. Mit POST können Ressourcen erstellt, mit PUT verändert und mit DELETE gelöscht werden.
Den Funktionen werden die Attibute Idempotenz und Sicherheit zugeordnet [6]. Eine Operation gilt als sicher, wenn sie aufgerufen werden kann, ohne den Zustand einer Ressource zu ändern. Ein Beispiel für eine sichere Operation ist GET. Da nur Repräsentationen abgerufen werden, darf sich der Ressourcen-Zustand nicht ändern. DELETE ist nicht sicher, da damit Ressourcen gelöscht werden können, was auch einer Änderung des Ressourcen-Zustandes gleichkommt. Idempotente Operationen erbringen immer das selbe Resultat, egal wie oft sie hintereinander aufgerufen werden. PUT ist idempotent. Eine Ressource wird lediglich verändert und mehrere Aufrufe sorgen für das gleiche Resultat. POST ist weder idempotent noch sicher. Sicherlich kann die Applikation auch so implementiert werden, dass

| Methode | sicher | idempotent | identifizierbare Ressource | Cache-fähig | sichtbare Semantik |
|---|---|---|---|---|---|
| GET | ✓ | ✓ | ✓ | ✓ | ✓ |
| POST | - | - | - | - | - |
| PUT | - | ✓ | ✓ | - | ✓ |
| DELETE | - | ✓ | ✓ | - | ✓ |

Tabelle II
EIGENSCHAFTEN VON FUNKTIONEN (VGL. „REST UND HTTP“) [9]

z.B. ein GET Seiteneffekte mit sich bringt und Ressourcen ändert. Da dies jedoch eine Verletzung der Forderung nach Sicherheit von GET darstellt, gilt eine solche Anwendung nicht als *RESTful*. Man stelle sich einen *Google-Crawler* vor, der beim alleinigen Abfragen von Repräsentationen Ressourcen löscht. [9]
*„In HTTP, a single resource-creating POST action will result in a 201 response with another hypertext representation (telling you what happened and what can be done next) or 204 response with the Location header field indicating the URI of the new resource. [11]“*
Operationen sollen standardisierte Resultate nach sich ziehen. Falls eine Ressource erfolgreich erstellt oder gelöscht wurde, soll dies dem *Client* mitgeteilt werden. Tabelle III zeigt einen kleinen Ausschnitt aus den HTTP-Status-Codes.

*Erweiterbarkeit, Skalierbarkeit und Wartung*

Durch den ressourcenzentrierten Ansatz ist ein *RESTful Web Service* ohne Änderung seiner API sehr effizient zu warten und zu erweitern. Durch das Hinzufügen neuer *Media Types* können neue *Clients* erschlossen werden ohne in die Logik der Applikation eingreifen zu müssen. Durch Bereitstellen weiterer *Links* in den jeweiligen Repräsentationen können andere *Services* mühelos integriert werden. Durch die zustandslose Kommunikation ist *Caching* einfach

| | | | |
|---|---|---|---|
| 200 | OK | 400 | Bad Request |
| 201 | Created | 401 | Unauthorized |
| 202 | Accepted | 403 | Forbidden |
| 204 | No Content | 404 | Not Found |
| 301 | Moved Permanently | 408 | Request Timeout |
| 302 | Found | 415 | Unsupported Media Type |
| 304 | Not Modified | 500 | Internal Server Error |
| 307 | Temporary Redirect | 501 | Not Implemented |

Tabelle III
EINIGE HTML-STATUS-CODES [12]

realisierbar und aufeinanderfolgende Daten müssen nicht zwingend vom gleichen System beantwortet werden. [13]

*Web Intermediaries*

Durch die Forderungen nach Zustandslosigkeit und der sinnvollen Nutzung von HTML-Standardmethoden können *Web Intermediaries* wie *Caches, Proxies* und *Firewalls* leicht auf verteilte Anwendungen angepasst werden bzw. diese unterstützen. Eine Operation muss in sich abgeschlossen sein. Damit können Abfragen mittels HTTP-GET sehr einfach in einem *Cache* vorgehalten werden. Auch der Zugriff lässt sich effizient regeln. Sämtliche Informationen, die das *Web Intermediary* benötigt, sind im HTTP-Header enthalten. Durch Herkunft und Zielressource der Anfrage kann der Zugriff sehr leicht gesteuert werden. Durch die *HTTP-Status-Codes* wird die Zugriffssteuerung sogar elegant auf Schnittstellenebene eingebettet. Die Nutzdaten müssen nicht betrachtet werden, was die Leistungsfähigkeit der Dienste unterstützt und dem Datenschutz entgegen kommt.

*Kompression*

HTTP unterstützt schon seit einiger Zeit Kompression. Vorausgesetzt die *Client-Anwendungen* unterstützen diese ebenfalls, kann so das Datenaufkommen reduziert werden. Kompressionsfunktionalität muss nicht extra in den *Web Service* implementiert werden.

*Sicherheit*

*„RESTful systems perform secure operations in the same way as any messaging protocol: either by encapsulating the message stream (SSL, TLS, SSH, IPsec) or by encrypting the messages (PGP, S/MIME, etc.). There are numerous examples of that in practice, and more in the future once browsers learn how to implement other authentication mechanisms.“* [11]
REST setzt keinerlei Grenzen oder Hindernisse bei der Integration von Sicherheitsmechanismen. Standardtechnologien wie Secure Sockets Layer (SSL), Pretty Good Privacy (PGP) oder OAuth sind durch den modularen, zustandslosen Aufbau problemlos einzubinden. Laut Statistik [3] finden HyperText Transfer Protocol Secure (HTTPS) und OAuth am häufigsten Verwendung.

*Dokumentation*

Durch die fehlende Methodik und Beschreibungssprache dokumentiert jeder *Service-Provider* auf seine eigene Art.
*„A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.“* Fielding fordert also, dass Dokumentation fast ausschließlich über die Medien Typen und durch die zur Verfügung stehenden *Links* bzw. Zustandsübergänge geschehen soll. Der *Client* soll durch die Applikation *geführt* werden. Überspezifikation in Form von Dokumentation fördert eine zu starke Kopplung von *Client* und *Server*. Da nur eine Hand voll festgelegter Zugriffsmethoden Verwendung findet, ist es nicht notwendig, diese zu dokumentieren. Viel wichtiger ist es in der Repräsentation anzugeben, welche Operationen auf einer Ressource zulässig sind und welche nicht.

## RESTFUL DESIGN

*„A truly RESTful API looks like hypertext. Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure). Query results are represented by a list of links with summary information, not by arrays of object representations (query is not a substitute for identification of resources). Resource representations are self-descriptive: the client does not need to know if a resource is OpenSocialist because it is just acting on the representations received.“* [11] - Roy T. Fielding
Ein guter Artikel zum Thema RESTful Design ist: „RESTful Web Services Development Checklist“von Steve Vinoski. Das Hauptaugenmerk beim Entwurf einer *RESTful* API sollte auf dem HATEOAS-Ansatz liegen. Ressourcen werden identifiziert und mittels URI addressiert. Gut gewählte Namen sind sinnvoll. Der *Service* muss frei von Zuständen sein. Nur Ressourcen dürfen Zustände annehmen. Als Repräsentation sollten vorhandene und allgemeine *Media Types* verwendet werden. Ist dies nicht möglich, können auch eigene *Media Types* entworfen werden. Gerade an diesem Punkt ist es wichtig, nicht nur die Nutzdaten der Applikation in JSON zu verpacken. Möglicherweise kann man ein Schema entwickeln, so dass eine Registrierung bei der IANA Sinn macht und auch andere Applikationen davon profitieren. Je nachdem wie viele *Clients* der *Service* bedient, sollte darüber nachgedacht werden, mehrere *Media Types* mittels *Content Negotiation* zur Verfügung zu stellen. Es sollte nur eine Einstiegs-Ressource in die Applikation geben. Repräsentationen dieser Ressource enthalten URIs, die die nächsten möglichen Zustandsübergänge des *Clients* darstellen. Dem Client muss die Möglichkeit gegeben werden, die Applikation frei zu durchlaufen. Die URIs sind komplett anzugegeben, um den *Client* nicht an ein URI-Schema zu binden. Datei-Endungen oder Parameter koppeln den *Client* zu stark an den *Server*. Die Nutzung von versteckten Methoden-Aufrufen in Parametern ist ein Indikator dafür, dass der Entwurf überarbeitet werden sollte. Von methodenorientiertem *Design* sollte sich generell gelöst werden. Auch sollte der Entwurf unabhängig vom verwendeten Protokoll gehalten werden. Die Zugriffsfunktionen sind sinnvoll und erwartungsgemäß zu implementieren. Des Weiteren ist bei der Implementierung auf Eigenschaften wie Idempotenz und Sicherheit zu achten. Tabelle IV zeigt eine Zuordnung von Zugriffs-Funktionen und Ressourcen-Typen. Der *Client* muss über *Status-Codes* über alle möglichen Anwendungsfälle informiert werden.

| | GET | PUT | POST | DELETE |
|---|---|---|---|---|
| /kontakt-formular | - | ✓ | ✓ | ✓ |
| /artikel | ✓ | - | - | - |
| /kunde | ✓ | ✓ | ✓ | - |
| /bestellung | ✓ | ? | ✓ | ? |
| /SOAP | - | - | ✓ | - |

Tabelle IV
HTML STANDARD-METHODEN [14]

## BEISPIEL AN HAND VON DOODLE

Doodle[5] ist ein bekannter *Web Service*. Möchte man mit seinen Kollegen eine Weihnachtsfeier veranstalten, ergibt sich meist das Problem der Terminfindung. Doodle ist ein *Web Service*, mit dem man relativ schnell und einfach Abstimmungen durchführen kann. Es wird ein *Web-Frontend* zur Benutzung im Browser und eine *RESTful* API [15] zur Entwicklung eigener *Clients* und *Mashups* bereitgestellt. Im Gegensatz zu vielen anderen *Web Services*, ist die Dokumentation einfach zu finden und überaus übersichtlich. Aufrufe werden genau erklärt, Parameter und Rückgabewerte spezifiziert. Selbst Beispiele werden angegeben. Das Datenformat ist XML und GET, PUT, POST und DELETE werden sinnvoll eingesetzt. Doodle verwendet nur drei verschiedene Ressourcen. *Polls*, *Participants* und *Comments*. Die XML-Schemata sind unter: http://doodle.com/xsd1 zu finden. Während man zu Testzwecken ohne Benutzer-Konto in einem separaten Bereich arbeiten kann, wird normalerweise Authentifizierung mittels OAuth verwendet.

Mit einer kleinen Umfrage über populäre Text-Editoren soll die Arbeit mit der Doodle API dargestellt werden. Zu Testzwecken wird auf der Doodle Homepage ein kleines Java-Programm bereitgestellt, mit der man relativ leicht HTTP-Funktionen absenden kann. Gerade zum Testen von *Web Services* hat allerdings das Programm cURL [6] bewährt.

Zu Beginn des Beispiels wurden *Poll-*[7] und *Participant-*[8] Schemata

```
<poll xmlns="http://doodle.com/xsd1">
  <type>TEXT</type>
  <hidden>false</hidden>
  <levels>2</levels>
  <title>Editor Battle</title>
  <description>tbd</description>
  <initiator>
    <name>John Doe</name>
  </initiator>
  <options>
    <option>Emacs</option>
    <option>Notepad</option>
    <option>Notepad++</option>
    <option>OpenOffice Writer</option>
    <option>UltraEdit</option>
    <option>Vi</option>
    <option>Vim</option>
    <option>Wordpad</option>
    <option>Word</option>
  </options>
</poll>
```

```
<participant xmlns="http://doodle.com/xsd1">
<name>Peter</name>
<preferences>
  <option>0</option>
  <option>0</option>
  <option>0</option>
  <option>0</option>
  <option>0</option>
  <option>0</option>
  <option>1</option>
  <option>0</option>
  <option>0</option>
</preferences>
</participant>
```

Abbildung 1. Umfrageschema und Umfragebeitrag

geladen und entsprechend angepasst. Abbildung 1 zeigt das ausgefüllte Umfrage-Formular. Anschließend wird in Listing 3 die Umfrage durch POST erstellt.

[5]http://www.doodle.com
[6]Client for URLs;http://curl.haxx.se/
[7]http://doodle.com/xsd1/poll.xsd
[8]http://doodle.com/xsd1/participant.xsd

```
curl.exe -v -POST -H "Content-Type: application/xml"
  -d @poll.xml http://doodle-test.com/
  api1WithoutAccessControl/polls
* About to connect() to doodle-test.com port 80 (#0)
*   Trying 217.197.135.77... connected
> POST /api1WithoutAccessControl/polls HTTP/1.1
> User-Agent: curl/7.22.0 (i386-pc-win32) libcurl/7.22.0
  zlib/1.2.5
> Host: doodle-test.com
> Accept: */*
> Content-Type: application/xml> Content-Length: 496
>
* upload completely sent off: 496out of 496 bytes
< HTTP/1.1 201 Created
...
< Location: http://doodle-test.com/api1WithoutAccessControl/polls/u65cirmaa6vry62t< Content-
Location: u65cirmaa6vry62t< X-DoodleKey: yhet7ub5
< Content-Length: 0
< Content-Type: text/plain
<
* Connection #0 to host doodle-test.com left intact
* Closing connection #0
```

Listing 3. Erstellung einer Umfrage

Aus dem Listing geht hervor, dass es sich um ein POST-Aufruf handelt, welcher Daten im Format application/xml an eine Umfrage-Ressourcen-Liste sendet, was typisch für die Erstellung von Ressourcen ist. Durch die Zieladresse ist hier gut zu erkennen, dass POST, nicht wie PUT auf einzelnen Ressourcen, sondern auf Mengen von Ressourcen operiert. Wichtig ist, dass die Operation sofort mit einem HTML Status-Code, in dem Fall *201 - Created*, quittiert wird. Ganz im Sinne des HATEOAS-Ansatzes wird eine URI auf die Umfrage selbst zurückgegeben, die man an Teilnehmer versenden oder abschließend, wie in Listing 4 dargestellt, abfragen kann. Der *Doodle-Key* ist nur in Kombination mit passwortgeschützten Umfragen relevant.

Zu bemängeln ist hier, dass ein Teilnehmer ohne Hintergrundwissen mit der URI nichts anfangen kann. Dass */participants* anzuhängen ist, geht nur aus der Dokumentation hervor.

```
curl.exe -v -GET -H "Content-Type: application/xml"
  http://doodle-test.com/api1WithoutAccessControl/
  polls/u65cirmaa6vry62t
* About to connect() to doodle-test.com port 80 (#0)
...
> Content-Type: application/xml
>
< HTTP/1.1 200 OK
...
< Content-Type: application/xml;charset=UTF-8
<
<?xml version="1.0" encoding="UTF-8"?>
<poll xmlns="http://doodle.com/xsd1">
  <latestChange>2011-12-18T22:23:48+01:00</latestChange>
  <type>TEXT</type>
...
  <initiator>
    <name>John Doe</name>
    <userId></userId>
    <eMailAddress></eMailAddress>
  </initiator>
  <options>
    <option>Emacs</option>
    <option>Notepad</option>
    <option>Notepad++</option>
    <option>OpenOffice Writer</option>
    <option>UltraEdit</option>
    <option>Vi</option>
    <option>Vim</option>
    <option>Wordpad</option>
    <option>Word</option>
  </options>
  <participants nrOf="3">
...
  </participants>
  <comments nrOf="0"></comments>
  <features></features>
</poll>
* Connection #0 to host doodle-test.com left intact
* Closing connection #0
```

Listing 4. Ergebnis einer Umfrage

```
curl.exe -v -POST -H "Content-Type: application/xml" -d
  @D:\tmp\wrk\SNET_Seminar\listings\opinion_peter.xml
  http://doodle-test.com/api1WithoutAccessControl/polls/
  u65cirmaa6vry62t/participants
* About to connect() to doodle-test.com port 80 (#0)
...
> Content-Type: application/xml
...
< HTTP/1.1 201 Created
...
* Connection #0 to host doodle-test.com left intact
* Closing connection #0
```

Listing 5. Erstellung eines Posts

```
curl.exe -v -X PUT -H "X-DoodleKey: yhet7ub5" -H
  "Content-Type: application/xml" -d
  @D:\tmp\wrk\SNET_Seminar\listings\opinion_peter_change.xml
  http://doodle-test.com/api1WithoutAccessControl/polls/
  u65cirmaa6vry62t/participants/14874
* About to connect() to doodle-test.com port 80 (#0)
...
> Content-Type: application/xml
...
< HTTP/1.1 200 OK
...
* Connection #0 to host doodle-test.com left intact
* Closing connection #0
```

Listing 6. Änderung eines Posts

```
curl.exe -v -X DELETE -H "X-DoodleKey: yhet7ub5"
  http://doodle-test.com/api1WithoutAccessControl/
  polls/u65cirmaa6vry62t/participants/14875
* About to connect() to doodle-test.com port 80 (#0)
...
< HTTP/1.1 204 No Content
...
* Connection #0 to host doodle-test.com left intact
* Closing connection #0
```

Listing 7. Löschen eines Posts

Mit dem neu gebildeten URI ist es möglich, Stimmen abzugeben (Listing 5), zu bearbeiten (Listing 6) und zu löschen (Listing 7). Die einzelnen Operationen, insbesondere ein fehlerhafter Versuch (Listing 8) eine Stimme zu ändern, werden durch verständliche Status-Codes quittiert.

```
curl.exe -v -POST -H "Content-Type: application/xml" -d
  @D:\tmp\wrk\SNET_Seminar\listings\opinion_missy.xml
  http://doodle-test.com/api1WithoutAccessControl/polls/
  u65cirmaa6vry62t/participants
* About to connect() to doodle-test.com port 80 (#0)
...
> Content-Type: application/xml
...
< HTTP/1.1 400 Bad Request
...
Input validation error(s) in business logic:
 * Item PARTICIPATION: INVALID (Reason: NOT_2LEVEL)
* Closing connection #0
```

Listing 8. Fehlerhafter Post

Implementierungstechnisch ist Doodle sicher kein besonders anspruchsvoller Service und es gibt mit großer Wahrscheinlichkeit Anwendungen, die sich deutlich schwerer *RESTful* abbilden lassen. Allerdings kann daran gut verdeutlicht werden, wie eine *RESTful* API aufgebaut werden sollte.

Anzumerken wäre, die anwendungsbedingt minimale Nutzung von URIs zwecks Selbst-Dokumentation. Es würde sich bei der Erstellung von Umfragen z.B. anbieten, Email-Adressen von Teilnehmern anzunehmen, um diesen anschließend eine Eintritts-URI zu senden. Das wiederum könnte als Seiteneffekt betrachtet werden, was auch nicht *RESTful* wäre. Ebenso könnte nach der Erstellung einer Umfrage ein Link auf eine Einladungs-Funktionalität bereitgestellt werden, über die die Teilnehmer explizit eingeladen werden könnten.

## Missverständnisse und Kritik des Authors

*„I am getting frustrated by the number of people calling any HTTP-based interface a REST API.“* [11] - Roy T. Fielding

Viele *Web Services* die angeblich *RESTful* implementiert sind, halten sich tatsächlich nur an einen Teil der Forderungen, die Fielding stellt. Die bloße Tatsache allein, dass eine Applikation in Ressourcen aufgeteilt und über eine hierarchische URI-Struktur bereitgestellt wird, macht noch keine *RESTful* API aus. Zum Teil werden Parameter in URIs verpackt, die sich auch in einem Schema selbst einbetten ließen. Noch schlimmer sind Parameter, die eigentlich Funktionsaufrufe verbergen, wie z.B. :

```
POST http://www.onlineshop.de/shop/bestellungen?bestellung_abschicken
```

Listing 9. RPC Parameter

Ein weiterer Kritikpunkt Fieldings ist, dass REST offenbar nur mit HTTP in Verbindung gebracht wird, obwohl gefordert wird, dass die Schnittstelle selbst, zwecks loser Kopplung, protokollunabhängig sein sollte. In der Praxis findet der Ansatz HATEOAS viel zu wenig Beachtung. Stattdessen wird versucht, mit unterschiedlichsten Ansätzen zu dokumentieren, da die Anwendung nicht selbsterklärend genug ist. Eine einheitliche Schnittstelle wird so nicht erreicht. Es sollten nur einige wenige Eintrittspunkte in die Applikation führen. Von dort aus sollten dem *Client*, ohne spezielle Vorkenntnisse, alle möglichen Zustandsübergänge mittels URIs angeboten werden. Dies müssen auch keine harten Links sein. Viel mehr sollten sie dynamisch erzeugt werden. Falls sich URIs ändern sollten, sollte dies vom *Server* durch *Status-Codes* oder Weiterleitungen behandelt werden. Wichtig ist auch, dass die URIs nicht vom *Client* zusammengesetzt werden müssen, da dies Hintergrundwissen voraussetzt und stark koppelt. Außerdem kann der *Server* so angepasst werden, ohne das es die *Clients* betrifft. Es sollte weniger Arbeit in die Ausarbeitung von URI-Hierarchien investiert werden. Viel wichtiger ist die Sorgfalt bei der Auswahl oder Entwicklung der *Media Types*. Daten lassen sich sehr schnell mittels JSON oder XML abbilden. Das verlockt natürlich für jede Applikation ein eigenes Schema zu erstellen, ohne vorher zu recherchieren, ob eventuell schon ein passendes Schema vorhanden ist. Vielleicht würde es sich auch anbieten, ein neues generisches Schema zu erstellen.

## Zusammenfassung

Fassen wir das Thema REST noch einmal zusammen: REST ist der Versuch, erfolgreiche Eigenschaften des *Webs* wie Skalierbarkeit, Erweiterbarkeit und *Interoperabilität* aufzugreifen und in einer Architektur zu vereinen.

Dafür stehen die Bedingungen: *Client-Server, Stateless, Cache, Uniform Interface, Layered System* und *Code on Demand*. Umgesetzt werden diese Bedingungen mit Hilfe von Ressourcen, deren Repräsentationen, HTML-Standard-Methoden und URIs. Wichtig bei der Gestaltung eines *Web Services* ist vor allem der ressourcenzentrierte Ansatz, der zusammen mit den Standard-Zugriffsmethoden eine einheitliche, übersichtliche und erweiterbare Schnittstelle schafft, so dass eine einfache Gestaltung autonomer- und nichtautonomer *Clients* und *Mashups* möglich ist. *Hypermedia* sollte den Antrieb des Applikationszustandes darstellen. Dies wird dadurch erreicht, dass Ressourcen massiv untereinander *verlinkt* werden, um eine intuitive und effiziente Art und Weise zu schaffen, über Standard-Methoden den Zustand des Clients zu ändern.

*Ressourcenorientiertes Design* und in sich abgeschlossene Operationen befriedigen die Forderung nach Zustandslosigkeit, was *Routing, Caching* und *Load Balancing* unterstützt. Die Forderung nach einer einheitlichen Schnittstelle ist der wohl größte Unterschied zwischen REST und SOA. Während die Schnittstelle durch REST fest und übersichtlich gehalten wird, kann sie im Bereich SOA, im Rahmen der Evolution einer Applikation, starken Änderungen unterliegen.

Die Welt der Web Services entwickelt sich sehr schnell. Während noch im Jahr 2010 auf *Programmable-Web* 2002 API und 4827 *Mashups* gelistet waren, waren es Ende 2011 schon 4678 API und 6362 *Mashups*. SOA ist im Bereich abgeschlossener und übersichtlicher

Systeme sicherlich eine gute Wahl. Obwohl an vielen angesprochenen Schwächen von SOA bereits gearbeitet wurde, ist meiner Meinung nach der ressourcenzentrierte Ansatz von REST für einen Großteil der verteilten Anwendungen besser geeignet.

Das *Web* ist heterogen und nicht abgeschlossen. Applikationen müssen leicht erweiterbar und wiederverwendbar sein, um an der Evolution des *Webs* beizutragen. Außerdem darf dies nicht nur für die Entwicklung innerhalb einer Gruppe gelten. Verteilte Systeme müssen ortsübergreifend entwickelt werden können. Wichtig ist auch, dass Anwendungen überhaupt erreicht werden können und keine Barrieren entstehen. Durch REST wird die Nutzung von Standards wie HTTP und HTML gefördert.

Allerdings zeigt die Studie von Maria Maleshkova, Carlos Pedrinaci und John Domingue [3], dass das Thema REST von vielen *Web-Entwicklern* falsch verstanden wird. Zu viele *Web Services* werden als *RESTful* angepriesen und halten nicht, was sie versprechen. Trotz Voraussetzungen wie *Uniform Interface* unterscheiden sich API enorm und es vergeht viel zu viel Zeit, sich mit der Bedienung vertraut zu machen.

REST ist sicherlich in der Definition sehr knapp gehalten. Einerseits aus zeitlichen Gründen, andererseits sind Architekturen generische Ansätze, damit sie überhaupt Anwendung finden können. Allerdings wird dadurch sehr viel Freiraum gelassen und es Neueinsteigern schwer gemacht, *Web-Anwendungen RESTful* zu gestalten. REST ist ein Ansatz verteilte Anwendungen möglichst langlebig und wiederverwendbar zu gestalten, was mit sehr hohem Aufwand und Erfahrung verbunden ist. Dies wird sicherlich von vielen *Designern* unterschätzt.

Nun sind seit *Fieldings Dissertation* gut zehn Jahre vergangen, trotzdem denke ich, dass REST erst in den letzten fünf Jahren wirklich populär geworden ist. In ein paar Jahren sind sicherlich viele Missverständnisse aus dem Weg geräumt und *Web Service* Entwickler haben aus ihren Fehlern gelernt. Was Zeit allerdings nicht wett machen kann sind die fehlenden Methodiken, Prozesse und *Tool-Unterstützung*, welche sich im Bereich SOA entwickelt haben. Vorhandene Ansätze, wie JAX-RS, müssen populärer gemacht werden. Außerdem sollten *Frameworks* beim *RESTful-Design* unterstützen und dafür sorgen, dass API und Dokumentation vereinheitlicht werden.

## LITERATUR


[1] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann, "Restful web services vs. big web services," 2008.

[2] S. Vinoski, "Restful web services development checklist," 2008.

[3] Maria Maleshkova, Carlos Pedrinaci, and John Domingue, "Investigatingweb apis on the worldwideweb," 2010.

[4] P. Prescod, "Second generation web services," *www.xml.com*, 2002.

[5] R. L. Costello, "Building web services the rest way," *www.xFront.com*, 2003.

[6] Leonard Richardson and Sam Ruby, *RESTful Web Services*. O'Reilly, 2007.

[7] R. L. Costello, "Representational state transfer," *www.xFront.com*, 2002.

[8] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, IRVINE, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000.

[9] S. Tilkov, *REST und HTTP*. dpunkt.verlag, 2011, vol. 2.

[10] *Uniform Resource Identifier (URI): Generic Syntax*, The Internet Engineering Task Force (IETF), http://www.ietf.org/rfc/rfc3986.txt, 2005.

[11] R. T. Fielding, "Rest apis must be hypertext-driven," http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008.

[12] Network Working Group, "Hypertext transfer protocol – http/1.1," http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, 1999.

[13] Thomas Bayer and Dirk M. Sohn, "Rest web services," *t3n*, 2007.

[14] C. Pautasso, "Restful service design," 2010.

[15] *RESTful Doodle*, http://doodle.com/xsd1/RESTfulDoodle.pdf.