Der bessere Web Service?

Vor nicht all zu langer Zeit schien es völlig klar, dass der Weg für eine interoperable Kommunikation zwischen Anwendungen mit unterschiedlichen Entwicklungszyklen über Web Services führen muss. In letzter Zeit jedoch ist immer häufiger die Rede von einer leichtgewichtigen, einfacheren Alternative: Web Services auf Basis von REST (REpresentational State Transfer).

von Stefan Tilkov

Die zugrunde liegende Behauptung dabei ist: Das Web ist gut genug, nur missverstanden. Macht man sich die Mühe, das Kernprotokoll HTTP näher zu betrachten, stellt man fest, dass das Web ohne WSDL, SOAP & Co. keineswegs weniger mächtig, sondern eher mächtiger ist. Mittlerweile ist REST einem ähnlichen Hype unterworfen wie SOAP, WSDL und UDDI vor einigen Jahren. Auch diverse Hersteller, allen voran Sun mit dem JSR 311/JAX-RS (siehe auch unser Beitrag auf Seite 82), sehen in REST zumindest einen wichtigen Teil der zukünftigen Technologielandschaft.

Viele Dienste im Web und viele Applikationen behaupten neuerdings, über eine „REST-Schnittstelle“ zu verfügen. Oft verbirgt sich dahinter nur der Verzicht auf einen SOAP Envelope oder ein banales Abbilden von Funktionen auf URLs. Was ist REST denn nun wirklich?

REST: Eine Definition

Der Begriff REST wurde von Roy T. Fielding, einem der Kernentwickler vieler Webstandards und ehemaligem Vorsitzenden der Apache Software Foundation, für seine Dissertation [1] eingeführt. Gemeint ist damit die Architektur, die den Webstandards, insbesondere dem HTTP-Protokoll, zugrunde liegt (theoretisch könnten auch andere Implementierungen dieser Architektur existieren, faktisch ist jedoch nur das Web mit HTTP, URIs usw. relevant). Die wesentlichen Elemente dieser Architektur sind:

- Ressourcen und Repräsentationen
- Hypermedia
- Einheitliche Schnittstelle

Eine Webanwendung ist mit Java-Bordmitteln – z.B. Servlets und dem ab Java 6 mitgelieferten eingebetteten HTTP-Server – leicht erstellt. Damit sie jedoch HTTP so verwendet, wie es dem REST-Architekturstil entspricht, muss sie sich an die Regeln halten, unabhängig davon, ob sie von einem Endanwender

per Webbrowser oder von einer anderen Anwendung als Service genutzt wird. Im Folgenden möchte ich Sie davon überzeugen, dass das Ergebnis den Namen „Web Service" sehr viel eher verdient als die WSDL/SOAP-Variante. Denn diese missachtet nahezu alle Grundsätze, die für den Erfolg des WWWs verantwortlich sind.

Ressourcen und Repräsentationen

Ressourcen sind das zentrale Konzept in REST; gelegentlich werden REST-Architekturen daher auch als „ROA" (Resource-oriented Architecture) bezeichnet. Alles, was es aus Sicht des Architekten verdient, eindeutig identifiziert zu werden, ist eine Ressource. Im Web wird für die Identifikation ein anwendungsübergreifend standardisierter, einheitlicher Mechanismus verwendet: den URI bzw. URL [2]. Generell gilt, dass man lieber zu viele als zu wenige Ressourcen identifizieren sollte. Erste Kandidaten sind die offensichtlichen Entitäten des Domänenmodells: Jede Bestellung, Filiale, Region, jedes Produkt, jeder Artikel kann mit einem eigenen URI identifiziert werden. Auch die „Liste aller Regionen, in denen der Umsatz größer als 5 Mio. Euro war", die „Produkte aus der Kategorie Spielwaren", alle „Bestellungen aus dem Jahr 2008" sind gültige Ressourcen und sollten einen möglichst stabilen und langlebigen URI erhalten.

Was ist der Vorteil eines solchen Entwurfs? Zum einen ist es die Möglichkeit, eine solche Ressource mit einem Lesezeichen im Browser zu verknüpfen oder einem Kollegen einen Link darauf schicken zu können. Zum anderen bietet das HTTP-Protokoll ausgefeilte Mechanismen für ein ressourcenbezogenes Caching, sodass in vielen Fällen Netzwerk-Requests vermieden oder minimiert werden können. Schließlich können Ressourcen mit unterschiedlichen URIs mit HTTP-Bordmitteln – z.B. mit einem Webserver oder einer Standardfirewall – feingranular unter-

schiedlich gegen unerlaubten Zugriff geschützt werden. Ressourcen in REST haben nicht nur eine, sondern potenziell mehrere Repräsentationen. HTTP unterstützt unterschiedliche Repräsentationen derselben Ressource über Content Negotiation (Accept- und Content Type Header). Eine Repräsentation ist eine Darstellung der Ressource, idealerweise in einem Standardformat. Unterschiedliche Klienten können damit jeweils das Format anfordern, das am besten ihren Bedürfnissen entspricht: Ein Browser eine HTML-Darstellung, ein programmatischer Client vielleicht eine XML- oder JSON-Repräsentation. Auch unterschiedliche Versionen – z.B. die Versionen 1.1 und 1.2 eines XML-Formats für die Darstellung einer Rechnung – lassen sich darüber abbilden.

Im Gegensatz dazu bilden SOAP/WSDL Web Services, aber auch viele Webanwendungen mit einer HTML-Benutzeroberfläche, ihre Funktionalität auf einen einzelnen URI ab. In einer Webanwendung führt das dazu, dass sich die Adresszeile im Browser nie ändert – ein Bookmark oder der Versand eines Links per E-Mail scheidet aus. Aber auch Suchmaschinen wie Google (oder äquivalente interne Dienste oder Appliances) können die in der Anwendung enthaltenen Konzepte nicht mehr erkennen. Bei einer REST-konform entworfenen Anwendung kann die Suchmaschine auf die Ressourcen der Anwendung zugreifen (und zwar nur in dem Maße, wie es die Anwendung zulassen möchte). Die Verwendung einer Web-Service-Schnittstelle dagegen erfordert Spezialkenntnisse bzw. ein genau dafür kodiertes Programm: an solchen Stellen ist das Web „zu Ende". Der SOAP- und WSDL- Begriff „Endpoint" passt hier sehr gut.

Hypermedia

Die Verbindung von identifizierbaren Ressourcen über Verknüpfungen (Links) macht das WWW zu einem Web: In den Repräsentationen von Ressourcen sind Links enthalten, die auf andere Ressourcen zeigen. Ein Client kann diesen Verknüpfungen folgen, und das standardisierte Identifikationsverfahren stellt sicher, dass sich die verknüpften Ressourcen in einem anderen Prozess, auf einem anderen Rechner im lokalen Netz oder auf der anderen Seite der Erde befinden können. Der Wert, den eine Anwendung dem weltweiten (oder auch firmeninternen) Netz hinzufügt, ist proportional zur Anzahl Ressourcen, die sie veröffentlicht. Ein REST-konform entworfenes CRM-System bei einem Telekommunikationsunternehmen z.B. würde Millionen von Kunden, Telefonnummern, Anschlüssen usw. als Ziel einer Verknüpfung verfügbar machen, sowohl für Endanwender als auch für Programme. Die Wahrscheinlichkeit einer Wiederverwendung steigt dramatisch.

Anwendungen können die Verknüpfungen entweder selbst auswerten oder dem Endanwender präsentieren. Idealerweise werden die nächsten möglichen Aktionen des Clients – die möglichen Statusübergänge – durch die in der Repräsentation enthaltenen Links vorgegeben. Neueinsteiger in die REST-Philosophie wenden oft eine erheblich Menge Zeit auf, um „schöne" URIs bzw. URI-Schemata zu erfinden. Klar strukturierte und menschlich lesbare URIs sind zwar besser als unlesbare, aber dieser Faktor wird häufig überbewertet. Viel wichtiger ist es, in der Kommunikation zwischen Client und Server die URIs als Teil des Inhalts der ausgetauschten Nachrichten zu übermitteln. Damit wird Hypermedia zum zentralen Konzept der Anwendungssteuerung; Fielding nennt dies „Hypermedia as the engine of application state". Auch wenn im SOAP- und WSDL-Universum mit dem WS-Addressing- Standard und den darin definierten Endpoint References mittlerweile auch eine Möglichkeit zur Identifikation von Objekten innerhalb einer Anwendung zur Verfügung steht, fehlt ein Konzept zur Verknüpfung. Natürlich könnte man URIs in SOAP-Nachrichten unterbringen – aber sobald sie verwendet (dereferenziert) werden, sind wir wieder in der Web/REST-Welt.

Einheitliche Schnittstelle

Vielleicht die ungewohnteste Restriktion des REST-Ansatzes ist die einheitliche („uniforme") Schnittstelle. Sie ist eine logische Konsequenz aus dem Anspruch, Anwendungen in ein Web von vernetzten Ressourcen zu transformieren. REST postuliert, dass jede Ressource die gleichen Operationen unterstützen muss. Im HTTP-Protokoll wird diese allgemeine Schnittstelle zu den bekannten Methoden GET, PUT, POST und DELETE (sowie u.a. den weniger bekannten HEAD und OPTIONS) konkretisiert. Natürlich verhalten sich Ressourcen unterschiedlich, aber sie haben einen gemeinsamen Kontrakt, der in der HTTP-Spezifikation [3] definiert wird. Eine spezifische Ressource, zum Beispiel eine Kundenliste, kann von einem darauf spezifisch zugeschnittenen Client genauso genutzt werden wie ein mit einer anderen Technologie realisierter Dienst (Abb.1). Implementiert die Ressource das HTTP-Interface korrekt, kann sie jedoch auch von einem generischen Client benutzt werden – sei es ein Browser, ein Kommandozeilenwerkzeug wie curl oder wget oder ein externer Dienst wie Google oder Yahoo!.

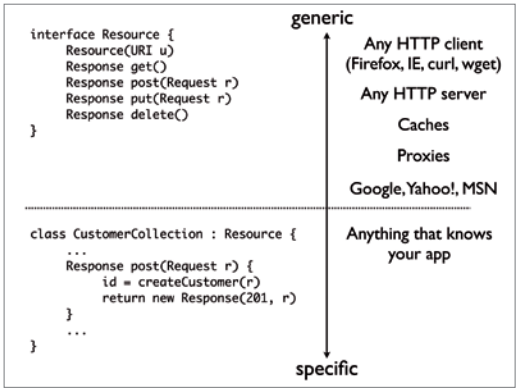

Abb. 1: Generische und spezifische Clients

Was genau legt die HTTP-Spezifikation als Semantik der einzelnen Methoden fest? Da die generische Schnittstelle auf alle Ressourcen anwendbar sein muss, ist sie naturgemäß allgemein gehalten. Sie definiert GET als eine lesende Operation, die „sicher" ist. Das heißt, dass ein Client durch ein HTTP GET keine Verpflichtung eingeht. Für GET ist außerdem ein sehr mächtiger Caching-Mechanismus spezifiziert, der eine feingranulare Kontrolle erlaubt. PUT legt eine neue Ressource unter einem bekannten Namen an bzw. aktualisiert sie dort. Ebenso wie DELETE und GET ist

Funktionalität	HTTP-Verb	URI	Format
Vorlage anlegen	POST	http://example.org/templates	Template
Adresse aktualisieren	PUT	http://example.org/addresses/4711	VCard
Adresse abfragen	GET	http://example.org/addresses/4711	VCard
Versandauftrag anlegen	POST	http://example.org/letters	Letter

Tabelle 1: Beispiel-URIs und -Methoden für das Serienbriefbeispiel

getFreeTimeSlots(Person)	→GET /people/{id}/timeslots?state=free
rejectApplication(Application)	→POST /rejections↵ <application>http://…</application>↵ <reason>Unsuitable for us!</reason>
performTariffCalculation(Data)	→POST /calculations↵ Data ←Location: http://…/calculations/4711 →GET /calculations/4711 ←Result
shipOrder(ID)	→PUT /orders/0815↵ <status>shipped</status>
shipOrder(ID) [Variante]	→POST /shipments↵ Data ←Location: http://…/shipments/4711

Tabelle 2: Abbildungen spezifische → generische Schnittstelle

PUT idempotent: Weiß der Client nicht, ob die Operation erfolgreich war oder nicht, kann er sie gefahrlos wiederholen. Ein POST legt eine untergeordnete Ressource an und ist die einzige Operation, für die es keine Garantien gibt.

Die Herausforderung für den Entwickler einer REST-konformen Anwendung besteht vor allem darin, die auf andere Architekturen wie CORBA, RMI, DCOM oder auch Web Services zugeschnittenen Gewohnheiten abzulegen. Es wird nicht für jeden Service eine eigene Schnittstelle entworfen, mit ganz spezifischen Operationen und Datentypen. Stattdessen muss die applikationsspezifische Funktionalität auf die uniforme Schnittstelle abgebildet werden. Im Gegensatz dazu wird bei SOAP/WSDL – ähnlich wie bei CORBA, DCOM und anderen Vorgängern – für jede Applikation eine neue, eben applikationsspezifische Schnittstelle erfunden. Dementsprechend können auch nur solche Systeme diese Schnittstelle nutzen, die explizit dafür entwickelt wurden.

Beispiel: Serienbriefe

Sehen wir uns zur Verdeutlichung ein konkretes Beispiel an: den Entwurf einer Schnittstelle für einen Serienbriefservice, einen Dienst für den Versand eines einheitlichen, ggf. personalisierten Texts an eine Vielzahl von Empfängern. In einem klassischen Entwurf würden Sie vielleicht zunächst die dienstspezifische Schnittstelle entwerfen, mit Operationen wie *Vorlage anlegen*, *Brief versenden*, Methoden zur Verwaltung von Adressen usw. Bei näherer Betrachtung ist diese Schnittstelle jedoch gar nicht so spezifisch wie man zunächst glaubt: Viele der zentralen Objekte werden vom Dienst (oder dem System, das den Dienst anbietet – je nach Terminologievorliebe) verwaltet, müssen also dort angelegt, geändert, gelöscht und ausgelesen werden. Diese CRUD-Funktionalität ist recht leicht abzubilden. Bei der REST- Variante unseres Beispieldienstes würden wir dazu vielleicht URIs der in Tabelle 1 dargestellten Form definieren.

Wir haben damit zunächst eine Art Stammdatenverwaltung für die Serienbriefe erstellt, die allerdings nun aufgrund der generischen Schnittstelle nicht nur von einem spezifischen Client, sondern auch von einem allgemeingültigen wie z.B. dem Browser verwendet werden kann. Voraussetzung dafür ist, dass das Format, der Content Type, bekannt ist – kleinster gemeinsamer Nenner ist HTML oder XML, aber auch ganz spezifische Formate wie z.B. VCard, eine standardisierte XML-Instanz, RSS oder Atom können sinnvoll sein. Und da eine Ressource mehrere Repräsentationen haben kann, lassen sich diese sinnvoll kombinieren – ein Standardformat für den generischen, ein spezifisches für einen genau auf die Anwendung zugeschnittenen Client.

Darüber hinaus müssen wir nun alle Operationen auf eine der Standardoperationen abbilden, die für Ressourcen zur Verfügung stehen. Aus der Operation *Brief versenden* wird damit *Versandauftrag anlegen*. Dieser Auftrag hat nun eine eindeutige ID (URI) und kann als Link-Ziel dienen, sein Status kann abgefragt werden, auch eine Änderung ist möglich (wenn auch sicher nur so lange, wie er noch nicht ausgeführt wurde). Die eigentliche Briefproduktionsmaschinerie kann ihre Aufträge abfragen, indem sie sich per GET über die Liste der noch offenen Aufträge informiert und den jeweils nächsten anstehenden verarbeitet. Neu angelegte oder abgeschlossene Aufträge könnten gleichzeitig als Newsfeed zur Verfügung gestellt werden, sodass man sich mit einem Standardnewsreader über den aktuellen Status informieren kann.

Die zentralen Konzepte – oder, wenn Sie möchten, Objekte – unserer Anwendung haben nun einen eigenen, stabilen URI. Diese Eigenschaft machen wir uns zunutze, um die erweiterte Funktionalität umzusetzen: Das Anlegen eines neuen Versandauftrags könnte aus einem POST von Links auf die Adressen, an die versandt werden soll, sowie einer weiteren Verknüpfung zur Vorlage bestehen. Wird ein Brief an eine große Anzahl von Empfängern versandt, könnte stattdessen ein Link auf die Collection-Ressource, die die passende Menge zurückliefert, enthalten sein. Das Spannendste daran ist, dass die einzelnen Bestandteile der Anwendung ebenfalls nur über die uniforme Schnittstelle gekoppelt sind. Eine andere Adressverwaltung als die in unsere Lösung integrierte könnte einfach angebunden werden, wenn sie ebenfalls einzelne URIs für ihre Elemente und den gleichen Content-Typ unterstützt.

Noch Fragen?

Ist REST nun der nächste große Hype nach Web Services? In gewisser Weise ist

die Antwort ein klares „Jein". Auf der einen Seite stammt die REST-Dissertation aus dem Jahr 2000, und sie beschreibt die Prinzipien, die dem Design des WWW schon vorher zugrunde lagen. Mit den Worten von Roy Fielding: „SOAP was known to be a bad idea in 1999, but in spite of our comments to this effect, the industry insisted on proving that for themselves". REST gab es also schon lange vor Web Services. Auf der anderen Seite ist unbestreitbar, dass aus einem Thema, das vor zwei oder drei Jahren noch relativ exotisch war, mittlerweile ein Aufmerksamkeitsmagnet geworden ist. Unabhängig stellt man sich nach einer ersten Beschäftigung mit dem Thema zunächst einige Fragen – irgendwie kann das Ganze nicht funktionieren! Oder doch? Die häufigsten Zweifel möchte ich im Folgenden ansprechen.

REST = CRUD?

Weil REST die Menge von Operationen auf einige wenige (GET, PUT, POST, DELETE) beschränkt, könnte man annehmen, mit REST könne man bestenfalls eine simple Datenverwaltung aufbauen, nicht jedoch fortgeschrittene Anwendungsfälle. Aber REST ist nicht das Gleiche wie CRUD: Ob es sich um eine Web-Service-Operation mit dem Namen *submitOrder* oder ein POST auf */orders* handelt: in beiden Fällen kann und sollte dahinter Geschäftslogik ablaufen und nicht nur ein simples Einfügen eines Datensatzes. Viele Operationen bewirken als Haupteffekt die Änderung oder Neuanlage eines Geschäftsobjekts; für diese Fälle ist die Abbildung auf das generische REST-Interface sehr einfach. Für die zusätzlichen Anwendungsfälle besteht der Trick darin, Aktionen auf eigene Ressourcen abzubilden: aus Verben werden Substantive. Statt einen Status zu ändern, legt man eine neue Statusänderung an, aus *cancel subscription* wird *create SubscriptionCancellation* usw. Einige Beispiele für eine solche Abbildung sind in Tabelle 2 dargestellt. Es erfordert zwar eine gewisse Übung; hat man sich jedoch einmal daran gewöhnt, denkt man ganz automatisch in Ressourcen. Darüber hinaus ergeben sich diverse Vorteile – individuelle Aktionen (bzw. ihre Ergebnisse) sind historisierbar, man kann sie verlinken, sie wieder zurücknehmen und die Ergebnisse in unterschiedlichen Formaten zur Verfügung stellen.

Formale Schnittstellenbeschreibung

Ein weiterer, häufig vorgebrachter Einwand: RESTful HTTP fehlt eine Analogie zu WSDL – die Schnittstellen sind nicht formal beschrieben. Das stimmt nur zum Teil: Eine Web-Service-Schnittstellenbeschreibung besteht zu einem erheblichen Anteil aus einer XML-Schema-Definition für die Nachrichteninhalte, und XML Schema ist selbstverständlich auch ohne WSDL und damit in REST-Anwendungen nutzbar. Was WSDL vor allem hinzufügt, ist die Benennung der einzelnen Operationen. Bei RESTful HTTP sind diese mit GET, PUT, POST, DELETE in der HTTP-Spezifikation selbst vorgegeben. Einer der Hauptzwecke einer WSDL (die Generierung von Code) ist für den wesentlichen Teil (die Daten) damit von WSDL unabhängig und auch im REST-Umfeld möglich. (Zweifel am Nutzen der automatischen Generierung von Code für die XML-Verarbeitung sind angebracht, aber das ist ein anderes Thema [4].) Zum anderen ist der Zweck der Schnittstellendokumentation auch mit WSDL nur sehr eingeschränkt gelöst: Diese wird in der Regel durch eine Prosadokumentation in Word, HTML oder PDF begleitet, weil die WSDL-Datei allein eben nicht ausreicht, um die Semantik der Schnittstelle zu verstehen. Beim REST-Ansatz bietet es sich an, Schnittstellenbeschreibungen im HTML-Format direkt mit den Ressourcen zu verknüpfen.

Unabhängigkeit vom Transportprotokoll

Diese vorgebliche Stärke von SOAP/WSDL Web Services ist tatsächlich eine Schwäche. Es hört sich hervorragend

an, unterschiedliche Transportprotokolle einsetzen zu können: HTTP oder direkt TCP, SMTP und POP3 oder auch das native Transportprotokoll einer JMS-Implementierung. Die Kehrseite ist jedoch, dass man nur den kleinsten gemeinsamen Nenner all dieser Protokolle nutzen kann. Analog könnte man einen Persistenzmechanismus definieren, der Daten im Dateisystem, in einem RDBMS oder in einem LDAP Directory ablegen kann: das ist zwar möglich – aber wie geht man z.B. mit Abfragen um? Entweder man verzichtet auf die Möglichkeiten der spezifischen Implementierung (z.B. auf SQL-Abfragen bei einem RDBMS) oder man entwickelt große Teile noch einmal auf höherer Abstraktionsebene neu (z.B. durch eine eigene Abfragesprache). Genauso steht es um das Verhältnis von Web Services und HTTP: Letzteres bietet sehr viel mehr als nur die Möglichkeit zum Transport von Bytes von A nach B; das ist schließlich die Aufgabe des darunterliegenden TCP-Protokolls. HTTP ist ein Applikationsprotokoll, das von Web Services gar nicht genutzt werden darf, ohne das Prinzip der Protokollunabhängigkeit zu durchbrechen. Letztlich tauscht man bei Einsatz der WS-*-Technologien einen mächtigen, etablierten, standardisierten und erprobten Protokoll-Stack (den des Webs) gegen einen anderen aus – und verzichtet dabei auf die zentralen Erfolgselemente wie URIs und Links.

REST für simple, WS-* für komplexe Fälle?

Nach einer ernsthaften Beschäftigung mit dem REST-Ansatz kommen viele Architekten und Entwickler oft zu dem Ergebnis, dass sich REST für einfache Fälle gut eignet. Für komplexere Enterprise-Szenarien aber wird der Einsatz von Web Services und insbesondere den erweiterten Spezifikation wie WS-Addressing, WS-Reliable Messaging, WS-Security usw. für notwendig gehalten. Dem lassen sich zwei Argumente entgegensetzen: Zum einen ist die Verbreitung eben dieser fortgeschrittenen WS-Standards in der Praxis äußerst gering: Die wenigsten Unternehmen riskieren die mit WS-Addressing und WS-Reliable Messaging verbundenen Interoperabilitätsprobleme oder akzeptieren den Performancenachteil nachrichtenorientierter Verschlüsselung. Zum anderen lassen sich auch fortgeschrittene Anwendungsfälle auf RESTful HTTP abbilden. Als ein Beispiel sei die zuverlässige Zustellung von Nachrichten genannt. Wie geht man damit um, wenn man auf eine Nachricht, einen Request, den man versandt hat, keine Antwort erhält? Schließlich weiß man nicht, ob der Request nicht angekommen, oder die Antwort darauf auf dem Rückweg verloren gegangen ist. HTTP begegnet diesem Problem damit, dass die Methoden GET, PUT und DELETE – wie bereits erwähnt – idempotent sind, d.h., wird ein solcher Request ein zweites Mal gesendet, ignoriert der Empfänger eine doppelte Zustellung einfach. Hat der Client auf eine Anfrage, also eine Antwort erhalten (bei der es sich auch nur um eine Eingangsbestätigung handeln kann), ist sie offensichtlich angekommen. Ansonsten sendet er sie einfach noch einmal. Ähnliche Muster gibt es für Anwendungsfälle wie Transaktionen, asynchrone Verarbeitung, Callbacks und viele andere „Enterprise"-Anwendungsfälle.

Fazit

Ressourcenorientierung, die uniforme Schnittstelle und Hypermedia – diese drei Elemente sind die zentralen Konzepte von REST. Eine Architektur, die diese Vorgaben umsetzt, unterscheidet sich deutlich von anderen Ansätzen. Schnittstellen, die als „REST-Interface" oder „REST-API" bezeichnet werden, halten sich oft nicht an die genannten Regeln. So „tunneln" sie z.B. die gesamte Funktionalität durch den Versand von XML-Dokumenten per POST oder bilden auch Methodenaufrufe, die Ressourcen ändern, auf ein GET ab. Mit anderen Worten: nicht überall, wo REST draufsteht, ist auch REST drin. Ist REST ein Allheilmittel? Natürlich nicht: es gibt zahlreiche Fälle, in denen z.B. asynchrones Messaging auf Basis eines MOM-Systems oder eine effiziente binäre Kommunikation auf TCP- oder UDP-Basis eine bessere Wahl ist. Aber wenn die Liste Ihrer Architekturziele Elemente wie lose Kopplung, Interoperabilität, Plattformunabhängigkeit, Skalierbarkeit und Internetfähigkeit enthält, ist RESTful HTTP aller Wahrscheinlichkeit nach die beste Wahl. Die Schnittstelle einer Anwendung zur Außenwelt nach REST-Prinzipien auf Basis von HTTP, URIs usw. zu entwerfen ist nicht einfach, auch wenn dies häufig behauptet wird. Die Herausforderung liegt vor allem im Umdenken – die Entwurfsansätze und -muster sind häufig neu und ungewohnt. Das Ergebnis jedoch ist ein System, das sich erheblich leichter in neuen und unerwarteten Kontexten verwenden lässt. Dem Ziel lose gekoppelter Systeme, einem Kerndogma von SOA, kommt man mit REST deutlich näher als mit althergebrachten Ansätzen – zu denen sich auch Web Services auf SOAP- und WSDL-Basis zählen lassen. ■

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, wo er sich mit serviceorientierten Architekturen auf Basis von Web Services und REST beschäftigt. Sie erreichen ihn unter stefan.tilkov@innoq.com.

Links & Literatur

[1] Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000: www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[2] RFC 2616: Hypertext Transfer Protocol 1.1: www.ietf.org/rfc/rfc2616.txt

[3] RFC 3986: Uniform Resource Identifier (URI): Generic Syntax: www.ietf.org/rfc/rfc3986.txt

[4] „Rethinking the Java SOAP stack", Loughran & Smith: www.hpl.hp.com/techreports/2005/HPL-2005-83.pdf