mehr zum thema:
www.innoq.com/resources/REST

REST:
DIE ARCHITEKTUR DES WEB

In diesem Artikel werden die Prinzipien erläutert, die der Architektur des WWW zu Grunde liegen. Anhand eines Beispiels wird gezeigt, dass diese nicht nur für die Kommunikation zwischen Web-Browser und -Server Anwendung finden können, sondern auch für die Kommunikation zwischen beliebigen anderen Systemen geeignet sind. Im Gegensatz zu Ansätzen, die dazu die Technologien aus der verteilten Objektwelt verwenden, beschränken wir uns dabei auf das oftmals unterschätzte HTTP-Protokoll.

Wer sich an die Zeit erinnern kann, in der noch nicht jede Pizzeria eine eigene Web-Präsenz hatte, der möge sich an folgendem Gedankenexperiment versuchen: Wir befinden uns in der Blütezeit der Client/Server-Welle und ein Bekannter stellt Ihnen seine Idee vor: Man könne doch einfach ein paar Standardprotokolle definieren, mit deren Hilfe Clients, die in beliebigen Programmiersprachen realisiert wurden und die auf unterschiedlichen Betriebssystemen laufen, mit jedem beliebigen Server kommunizieren können, ebenfalls unabhängig von deren Implementierungsplattform. Einfach eine gigantische, weltweit installierte Anwendung, in der Elemente universell adressierbar und damit referenzierbar sind, dabei lose gekoppelt, offen, hinreichend performant mit eingebauten Mechanismen für Lastverteilung, Indirektion, Umleitungen und mit standardisierten Fehlermeldungen – und das alles herstellerübergreifend und kostengünstig.

Wahrscheinlich würden Sie den Menschen, der Ihnen diese Idee präsentiert, für einen naiven Idealisten halten, der durch die harte Realität noch früh genug auf den Boden der Tatsachen zurückgeholt werden wird. Tatsache ist jedoch, dass das World Wide Web unbestritten die „Killerapplikation" für das Internet geworden ist und Tim Berners-Lees realisierte Vision mittlerweile aus unserem Alltag nicht mehr wegzudenken ist.

In diesem Artikel wollen wir zeigen, wie sich REST, die Architektur des WWW, für die Anwendungs-zu-Anwendungskommunikation einsetzen lässt. *REST* ist die etwas gewöhnungsbedürftige Abkürzung für die noch gewöhnungsbedürftigere Langform *REpresentational State Transfer*. Roy Fielding hat in seiner Dissertation die Architektur des Web charakterisiert und den Namen REST geprägt (vgl. [Fie00]).

Anwendungsbeispiel: Bestellverwaltung

Wir möchten die Grundprinzipien von REST und deren Anwendung an einem durchgehenden Beispiel erläutern. Dazu haben wir uns entschieden, eine einfache Bestellverwaltung REST-basiert zu konzipieren. Zur Erläuterung des Beispiels beginnen wir klassisch mit einer objektorientierten Analyse, um die Bestellverwaltung zu charakterisieren. Im Folgenden transformieren wir dann die Ergebnisse auf die Grundprinzipien von REST und Konzepte unserer REST-basierten Architektur. Wir hätten das Beispiel auch direkt mit einer an REST angelehnten Vorgehensweise analysieren können, wollten aber einen klassischen Einstieg, um von Bekanntem zu Neuem überzugehen.

Ausgehend von einer einfachen Analyse der möglichen Anwendungsfälle soll unsere Beispielanwendung die in Abbildung 1 dargestellten Anwendungsfälle kennen. Ein erster objektorientierter Entwurf eines solchen Systems könnte für jeden Anwendungsfall eine Operation innerhalb eines Service exportieren (siehe Abb. 2). Die Operationen werden in zwei wesentlichen Services zusammengefasst: BestellService und KundenVerwaltung. Offensichtlich benötigen wir zwei nicht weiter ausgeführte Geschäftsobjekte: Bestellung und Kundenakte.

Versuchen wir nun, das System REST-basiert zu realisieren. Dazu betrachten wir zuerst die Grundprinzipien von REST (siehe auch Kasten 1).

Grundprinzipien des REST-Stils

Vereinfacht gesagt ist REST eine Architektur für verteilte Anwendungen, die von den Kernprinzipien des WWW-Protokolls, HTTP, technologieunabhängig abstrahiert. Zu den Prinzipien des REST-Architekturstils gehören die zustandslose Client/Server-Kommunikation, identifizierbare Ressourcen, Ressourcen-Repräsentationen, die uniforme Schnittstelle sowie Hypermedia (Verwendung von Links).

Auch wenn uns die Entscheidung für REST nicht die Verwendung von HTTP aufzwingt, wollen wir in unserem Beispiel HTTP als Applikationsprotokoll nutzen. Die Verwendung von HTTP allein garantiert allerdings nicht, dass unsere Lösung REST-konform ist. Vielmehr müssen wir unseren Entwurf sorgfältig gegen die Grundprinzipien prüfen, um sicherzustellen, dass zum Beispiel kein Konversationszustand gehalten werden muss. Betrachten wir die Grundprinzipien im Detail.

Zustandslose Kommunikation

Die Kommunikation zwischen Client und Server erfolgt im REST-Modell grundsätzlich zustandslos, d. h. der Client kann sich bei der Kommunikation nicht darauf verlassen, dass ein in einer vorherigen Interaktion entstandener Kontext immer noch vorhanden ist. Stattdessen müssen

die autoren

Stefan Tilkov
(E-Mail: stefan.tilkov@innoq.com) ist Geschäftsführer und Principal Consultant der innoQ Deutschland GmbH. Dort beschäftigt er sich schwerpunktmäßig mit modellgetriebenen Software-Entwicklungsansätzen und service-orientierten IT-Architekturen.

Phillip Ghadir
(E-Mail: phillip.ghadir@innoq.com), Principal Consultant und Mitglied der Geschäftsleitung bei innoQ, beschäftigt sich seit vielen Jahren mit effektiven Methoden zur rationellen Softwareproduktion und Architekturansätzen für verteilte, kritische, unternehmensweite Systeme.

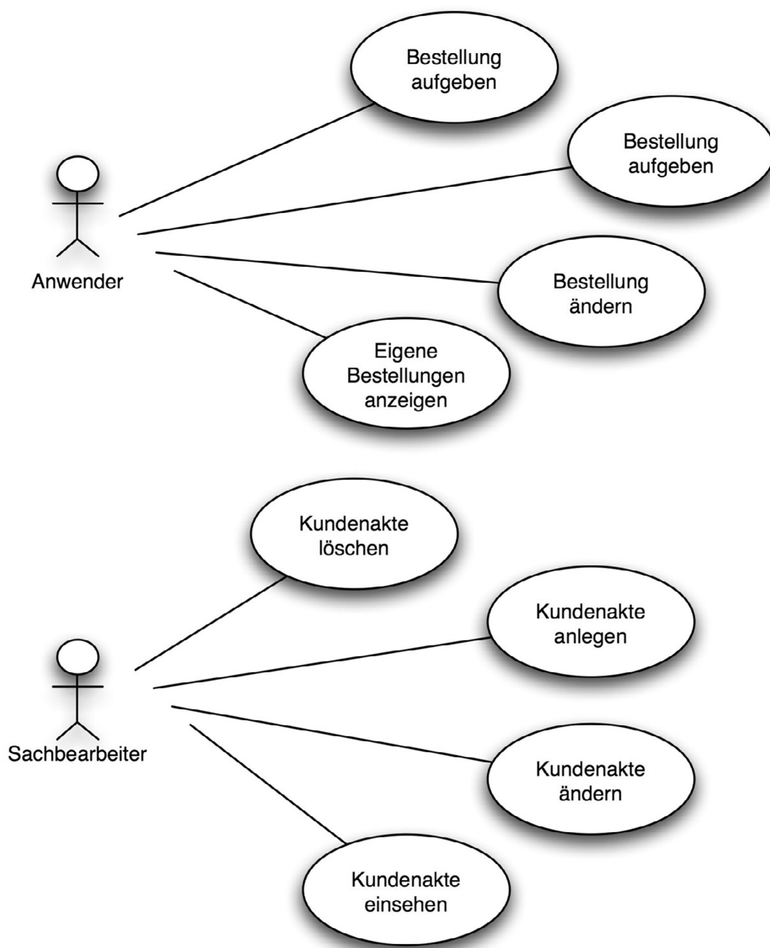

Abb. 1: Anwendungsfälle der simplen Bestellverwaltung

alle Informationen, die der Server zur Verarbeitung einer Anfrage benötigt, in dieser Anfrage enthalten sein. Als Konsequenz daraus muss der Server nicht für jede aktuell laufende Interaktion möglicherweise begrenzte Systemressourcen (wie z.B. Hauptspeicher, Datenbankverbindungen oder *Threads*) zur Verfügung stellen – woraus sich ein unmittelbarer Vorteil bei der Skalierbarkeit, Lastverteilung, Ausfallsicherheit usw. ergibt.

Die zustandslose Kommunikation ist ein wesentlicher Unterschied zu Ansätzen mit verteilten Objekten, wie etwa CORBA: Hier erhält der Client vom Server eine Objektreferenz, die er verwenden kann, um mehrere Methoden nacheinander aufzurufen. Dabei kann er sich jederzeit darauf verlassen, dass das zustandsbehaftete Objekt vom Server für ihn – und nur für ihn – reserviert ist.

Identifizierbare Ressourcen

Auch wenn man dem Prinzip der zustandslosen Kommunikation folgt, kann natürlich immer noch ein dauerhafter Zustand existieren. Dies ist bei REST nicht nur möglich, sondern wird sogar explizit durch das zweite wesentliche Grundprinzip unterstützt: durch identifizierbare Ressourcen. Eine Ressource ist in diesem Zusammenhang ein sehr allgemeiner Oberbegriff für konkrete oder abstrakte Dinge, mit denen eine Interaktion möglich ist. In unserem Beispiel der Bestellverwaltung kennen wir die Ressourcen *Bestellungen*, *Bestellung*, *Kunden* und *Kunde* (siehe Abb. 3). Sie repräsentieren Objekte bzw. Mengen von Objekten aus der wirklichen Welt. Allen Ressourcen ist gemeinsam, dass sie über einen standardisierten Mechanismus dauerhaft weltweit eindeutig identifizierbar und adressierbar sind. Eine Ressource kann ihre Daten auf Wunsch in einer beliebigen Anzahl unterschiedlicher Repräsentationen liefern, z.B. im HTML- oder XML-Format, als Grafik oder als PDF. Auf diese Eigenschaft – den Transfer von Ressourcen-Repräsentationen – geht der Name *REST (REpresentational State Transfer)* zurück.

Jede dieser Ressourcen ist in der Bestellverwaltung über einen eindeutigen URI referenzierbar – doch dazu später mehr.

Uniforme Schnittstellen

Das vielleicht bedeutendste und für Kenner anderer Architekturstile am schwersten als positiv zu akzeptierende REST-Prinzip ist das der gleichförmigen (uniformen) Schnittstelle. Im Gegensatz zu RPC-Ansätzen oder Ansätzen mit verteilten Objekten gibt es bei REST nur eine einzige Schnittstellenvereinbarung, die nur die Methoden bzw. Operationen beinhaltet, die prinzipiell von allen (!) Ressourcen unterstützt werden. Eine Analogie ist die Schnittstelle, die vom Datenbankstandard SQL für den Zugriff auf Daten zur Verfügung gestellt wird: Mit Hilfe der Grundbefehle INSERT, UPDATE, SELECT und DELETE lassen sich im Grunde alle Operationen, die beim Umgang mit der Datenbank notwendig sind, abbilden. Ein anderes Beispiel für eine gleichförmige Schnittstelle ist der aus der Unix-Welt stam-

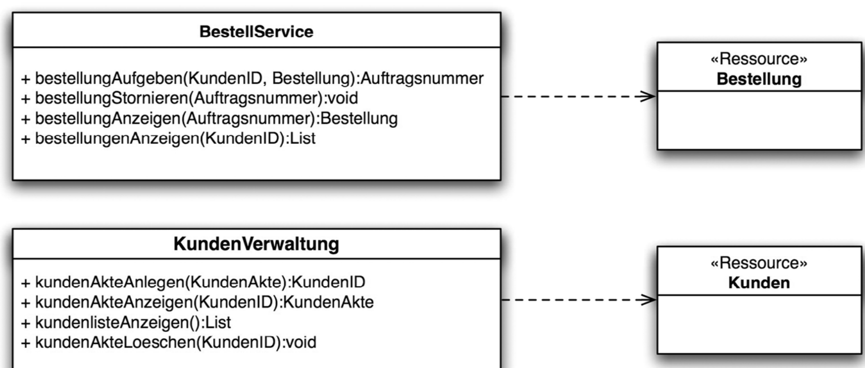

Abb. 2: Grobentwurf des BestellService: ein klassischer (nicht-REST-basierter) Entwurf

- zustandslose Client/Server-Kommunikation
- identifizierbare Ressourcen
- Ressourcen-Repräsentationen
- uniforme Schnittstelle
- Hypermedia (Verwendung von Links)

Kasten 1: *REST-Prinzipien*

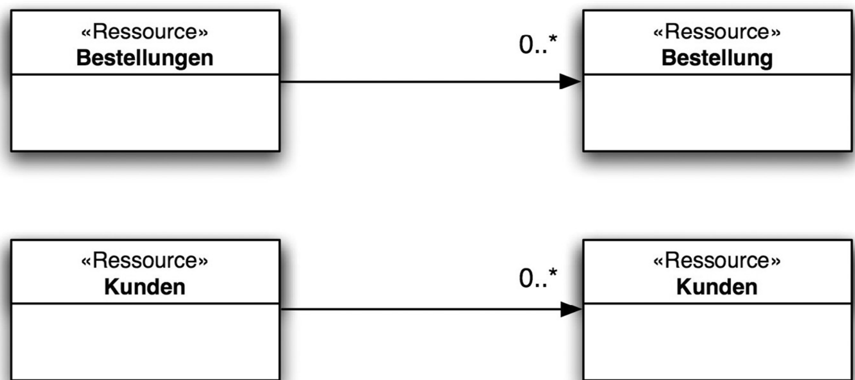

Abb. 3: *Identifizierte Ressourcen der Bestellverwaltung*

mende *Pipelining*-Mechanismus: Einfache Kommandozeilenwerkzeuge, die vom Standardeingabe-Kanal lesen und auf die Standardausgabe schreiben, lassen sich zu überaus mächtigen Prozessketten zusammenfassen, indem die Ausgabe des einen als Eingabe des jeweils nächsten Prozesses verwendet wird.

Der Vorteil gleichförmiger Schnittstellen ist ihre Universalität: Es gibt eine große Menge von Anwendungsfällen, die auf ihrer Basis generisch abbildbar sind. Das gilt für Applikationen wie den Web-Browser, der mit jeder beliebigen Website interagieren kann, ohne dafür jedes Mal eine neue Schnittstelle „erlernen" zu müssen, oder für Hilfswerkzeuge wie „curl", das wir in den nächsten Abschnitten noch näher kennen lernen werden.

Hypermedia

Das letzte Prinzip schließlich ist „Hypermedia as the Engine of Application State". Dahinter verbirgt sich das Modell, das wir vom Web-Browser kennen: Vom Server wird nicht nur eine Menge von Daten (Ressourcen-Repräsentationen in REST-Terminologie) übertragen, sondern auch die durch den Client initiierbaren Zustandsübergänge in Form von eingebetteten Verknüpfungen (*Links*). Der Server überträgt damit alle Informationen zum Client, die dieser benötigt, um entweder benutzergesteuert oder automatisiert den nächsten notwendigen Schritt einzuleiten. Die Bestellverwaltung ergänzt jede Ressourcen-Repräsentation um eine Liste mit URIs der assoziierten Ressourcen. Wird beispielsweise die Liste der Bestellungen angefragt, wird eine Liste von URIs auf die einzelnen Bestellungen sowie der URI zum Kunden zurückgeliefert.

Bestellverwaltung in REST

Die wichtigste Inkarnation des REST-Stils ist das WWW mit seinen Basisstandards HTTP, HTML, XML und URI. HTTP ist ein Applikationsprotokoll, das die Manipulation von Ressourcen unterstützt, die über URIs identifiziert werden. Der Datenaustausch erfolgt mit Hilfe von Ressourcen-Repräsentationen, die unterschiedliche Typen bzw. Formate haben können. Die Schnittstelle, über die die Kommunikation erfolgt, ist dabei einheitlich und vom konkreten Anwendungsfall unabhängig. Eine umfangreiche Menge von Statuscode wird für unterschiedliche Fehlerfälle bzw. Verarbeitungsstatus vorgegeben. All dies wollen wir nun für unsere Bestellverwaltung nutzen (einige Hinweise zur Vorgehensweise beim Design von REST-basierten Anwendungen finden Sie in Kasten 2).

Ressourcen und URIs

Das vielleicht sichtbarste im Web umgesetzte REST-Prinzip sind URIs. *URI* steht für *Uniform Ressource Identifier*, und genau darum geht es: die gleichförmige, einheitliche Identifikation von Ressourcen.[1])

Wir gehen täglich mit Web-Ressourcen um: der Link auf das Buch, das Sie Ihrer Kollegin empfehlen wollten, die Adresse des Hotels, in dem Sie ihren Urlaub verbringen, der vorkonfigurierte Warenkorb mit Ihrem Wunsch-Notebook. Wir sind es gewohnt, URIs in die Adresszeile unseres Browsers einzutippen und Links in den Dokumenten, die wir betrachten, zu folgen. URIs und die durch sie ermöglichten Links sind ein entscheidender Faktor für den Erfolg vieler Web-Anwendungen. So ist es sicher unstrittig, dass ein Unternehmen wie Amazon.com einen erheblichen Anteil seines Umsatzes der Tatsache verdankt, dass es viel praktischer ist, einen Link auf ein Buch per E-Mail zu versenden als die genauen Angaben über Autor und Titel. Es spricht einiges für die These, dass der „Wert" einer Anwendung mindestens proportional zu der Anzahl der Links steigt, die sie dem globalen Adressraum hinzufügt. Auch das Gegenargument ist schlüssig: Eine Anwendung, die zwar HTTP, aber nur einen einzigen URI verwendet, unterminiert das Prinzip von Links und verzichtet so auf ein unter Umständen enormes Potenzial.

In unserer Bestellverwaltung möchten wir für die Menge der Bestellungen eines jeden Kunden einen URI anbieten. Weiterhin soll jede Bestellung selbst über einen URI referenzierbar sein. Analoges gilt für jeden Kunden. Eine „Suche nach Kunden" ermöglichen wir in unserem Beispiel über die URI der Ressource Kunden, die um einschränkende Attribute ergänzt werden kann (siehe Abb. 4).

Somit ist es möglich, Links auf Bestellungen oder Kundenakten an die Mitarbeiter der Buchhaltung zu senden oder an entsprechende Listen von Marketing-Aktionen anzufügen, die zu einer Bestellung geführt hatten.

Bei der Entwicklung einer REST-konformen Anwendung ist daher die Modellierung der Ressourcen einer der ersten Schritte – ähnlich zur objektorientierten Analyse, bei der das Finden und Modellieren von Objekten einen der ersten

[1]) Wir verwenden den Begriff URI statt URL; die früher übliche und vielleicht dem einen oder anderen noch geläufige Unterscheidung zwischen URI, URL und URN ist nicht mehr relevant (vgl. [W3C01]). Wenn Sie den Eindruck vermitteln möchten, ganz besonders auf der Höhe der Zeit zu stehen, können Sie den Begriff „IRI" (Internationalized Ressource Identifier) verwenden. IRIs werden durch RFC 3987 (vgl. [Int]) definiert und sind im Prinzip URIs mit der Möglichkeit, alle Zeichen aus dem Unicode-Zeichensatz zu verwenden.

Für die Entwicklung von Anwendungen, die ihre Kommunikation per HTTP und möglichst konform zu den REST-Prinzipen abwickeln sollen, hat es sich bewährt, den folgenden Prinzipien zu folgen.

- Entscheiden Sie, welche Ressourcen in Ihrem System existieren und wie diese miteinander verknüpft sind. Lassen Sie sich dabei davon leiten, welche Dinge sie durch eine URI eindeutig identifizieren möchten – z. B. weil Sie sich vorstellen können, einen Link als Lesezeichen in Ihren Browser aufzunehmen oder per E-Mail zu versenden.
- Modellieren Sie *Collections* als Ressourcen, die eine Kurzbeschreibung der in ihnen enthaltenen Elemente beinhalten und – da es sich bei diesen auch wiederum um Ressourcen handelt – einen Link darauf.
- Vermeiden Sie es, der Struktur ihrer URIs zu viel Bedeutung beizumessen, insbesondere auf der Client-Seite. Es ist durchaus sinnvoll, „schöne" URIs zu entwerfen – z.B. ist http://example.org/orders/2005/12345.xml deutlich eleganter als http://example.org/AC23DA2A62EA. URIs sind jedoch vom Ansatz generell intransparent, d.h. solange der Server sie interpretieren kann, ist dem Client ihre Struktur gleichgültig. Eine *Collection* mit einer URI http://domain1/X, die nur Element mit URIs der Form http://domain1/X/y enthält, mag Ihnen heute vollkommen ausreichend erscheinen – denken Sie aber auch darüber nach, dass das Web Ihnen die Möglichkeit bietet, morgen eine Ressource http://otherdomain/whatever in Ihre *Collection* aufzunehmen. Idealerweise liefern Sie den Entwicklern, die Ihr REST-basiertes API verwenden sollen, nur eine einzige URI. Hinter dieser verbirgt sich ein XML-Dokument, in dem die nächsten erreichbaren Ressourcen in Form von URI-Links enthalten sind.
- Entscheiden Sie für die Ressourcen, die Sie gefunden haben, welche der HTTP-Methoden diese unterstützen und welche Semantik sich dahinter verbirgt. Beachten Sie dabei, dass die HTTP-Methoden bereits eine Semantik haben, gegen die Sie möglichst nicht verstoßen sollten. Ebenso müssen Sie definieren, welche Formate Ihre Ressourcen-Repräsentationen haben sollen und wie Sie mit den Statuscodes umgehen.
- Die Möglichkeiten, die das HTTP-Protokoll zur Kodierung und Aushandlung der Medientypen bietet, können Sie verwenden, um für Ihre Ressourcen neben der maschinell lesbaren XML-Form noch eine HTML-Darstellung zu liefern und so den Zugriff für Endanwender zu erleichtern – und sei es auch nur aus Debugging-Gründen.
- Vermeiden Sie *Cookies* und andere Tricks, die das bewusst statuslose HTTP-Protokoll in ein statusbehaftetes, konversationsorientiertes transformieren.

Kasten 2: *Entwurf von REST/HTTP-Anwendungen*

Schritte darstellt. In unserem Beispiel haben wir darauf geachtet, primäre Ressourcen zu finden: Kunden und Kunde, Bestellungen und Bestellung. Anstelle eines Bestellverwaltungsservice haben wir Listen von Ressourcen als geeignete Kandidaten ebenfalls zu Ressourcen gemacht: „Alle Bestellungen eines Kunden", „die Liste der Kunden". Diese *Collection Resources* beinhalten Links auf die primären Ressourcen.

Bevor wir auf die gleichförmige Schnittstelle und die HTTP-Verben eingehen, möchten wir noch ausdrücklich darauf hinweisen, dass wir die Bestellverwaltung mit der gewünschten Funktionalität REST-basiert entwerfen, ohne dabei speziell einen Service vorzusehen. Die Architektur des Web, gepaart mit den Ressourcen der Anwendungsdomäne „Bestellverwaltung", und unsere Art, die Ressourcen zu verknüpfen, reichen aus, um die Bestellverwaltung voll funktionsfähig und REST-basiert zu entwickeln.

Gleichförmige Schnittstelle und HTTP-Verben

Das REST-Prinzip der gleichförmigen, einheitlichen Schnittstelle wird im Web durch die so genannten Verben des HTTP-Protokolls umgesetzt; dies sind die universell unterstützten, standardisierten Operationen, die über das HTTP-Protokoll angestoßen werden können.[2]) HTTP 1.1 (vgl. [Net99]) definiert *acht Verben*; die wichtigsten vier davon sind:

- GET fordert eine Repräsentation einer Ressource an.
- PUT legt eine neue Ressource an oder ersetzt eine bestehende.
- POST übermittelt Daten zur Verarbeitung durch die angegebene Ressource.
- DELETE löscht eine Ressource.

Die klare Definition der Semantik der HTTP-Methoden und eines Satzes von Rückgabestatus macht HTTP zu einem echten Applikationsprotokoll und hebt es dadurch von reinen Transportprotokollen ab.

Auch für unser Beispiel möchten wir HTTP und die Vorteile nutzen, die sich aus der bestehenden Web-Infrastruktur ergeben. In unserem Design übernehmen wir die exakte Semantik der HTTP-Methoden. Die Methoden GET und HEAD sind *sicher* und *idempotent*. Sie haben keine Seiteneffekte und verändern auf dem Server keinen Zustand. In unserer Bestellverwaltung unterstützen wir nur GET und liefern damit Repräsentationen von Ressourcen zurück.

Die Methoden PUT und DELETE sind *idempotent*. Die an der Kommunikation beteiligten Komponenten dürfen gefahrlos noch einmal ausgeführt werden. Ein bereits gelöschtes Objekt per DELETE noch einmal zu löschen, ist genauso ungefährlich, wie eine Ressource über PUT mehrfach mit dem gleichen Inhalt zu aktualisieren.

Insbesondere in unserem Beispiel verwenden wir POST zum Anlegen einer neuen Ressource. Eine mehrfache Ausführung derselben Anfrage würde also unseren Datenbestand verändern. Deshalb fragen Browser bei der Aktualisierung von Seiten, die als Antwort auf POST geladen wurden, ob die Daten erneut übertragen werden sollen.

Wir verwenden die HTTP-Methoden exakt gemäß ihrer Bestimmung und sind

[2]) *Zu den hier genannten kommen noch die weniger relevanten, aber ebenfalls standardisierten Verben* TRACE, HEAD, OPTIONS *und* CONNECT *hinzu. Die Anzahl der Verben ist zwar prinzipiell erweiterbar, davon wird aber nur äußerst selten Gebrauch gemacht. Ein Beispiel dafür ist WebDAV, das HTTP-basierte Protokoll zur Verwaltung von entfernten Dateihierarchien.*

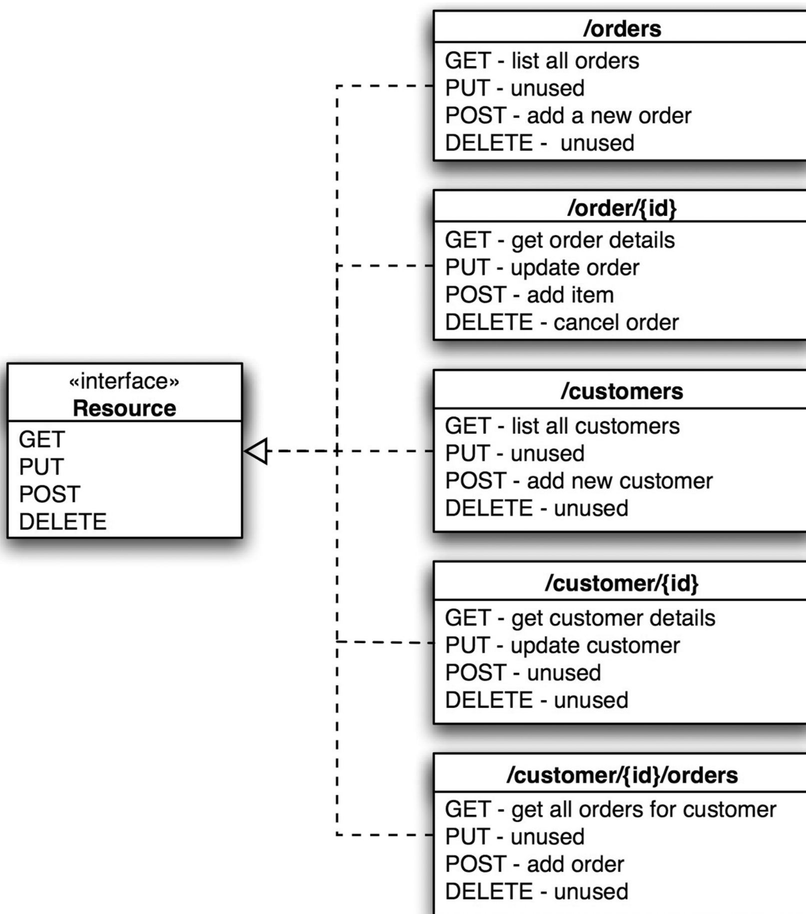

Abb. 4: Ressourcen, URIs und die Bedeutung der HTTP-Methoden

damit einfach in der Lage, Suchanfragen zu formulieren, Kundenakten anzulegen oder Bestellungen nachträglich zu ändern.

Ressourcen-Repräsentationen und MIME-Typen

Die Repräsentationen von Ressourcen, die beim Zugriff auf die URIs mit der entsprechenden Methode ausgetauscht werden, können unterschiedliche Formate haben. Bei einer GET-Anfrage gibt der Client in einem Accept-Header an, welche Typen von Inhalten (Content Types) er akzeptiert. Dabei kann eine Priorisierung mit angegeben werden.

Auch Antworten sowie PUT- und POST-Anfragen, die Inhalte transportieren, geben mit Hilfe des Content-Type-Headers an, welcher Repräsentationstyp tatsächlich gewählt wurde.

In unserer REST-konformen Beispiel-Bestellverwaltung ordnen wir unseren Ressourcen die folgenden Repräsentationstypen zu. Über die Mechanismen zur Aushandlung der unterstützten MIME-Typen (die Content Negotiation) können die Kommunikationspartner die geeignete Repräsentation dynamisch festlegen:

- ORDERS = text/plain, text/html, application/xml
- ORDER = text/plain
- CUSTOMERS = text/plain, text/html, application/xml
- CUSTOMER = text/plain, application/xml

Verwendung eines universellen Clients für die Bestellverwaltung

Wir haben uns zu Beginn bewusst entschieden, die REST-Architektur unserer Bestellverwaltung mit HTTP zu realisieren, um so die gesamte Infrastruktur des WWW sowie die bestehenden Softwarewerkzeuge verwenden zu können. Da die Bestellverwaltung in erster Linie für den Einsatz im Bereich der Anwendungs-zu-Anwendungskommunikation gedacht ist, haben wir uns für die (REST-konforme) Verwendung der HTTP-Methoden GET, PUT, POST und DELETE entschieden. Nun ermöglicht es uns ein Web-Browser, nur GET- oder POST-Anfragen zu senden, sodass wir hier einen anderen universellen Client bemühen, um die Funktionsweise unserer Anwendung zu testen: das kleine Unix-Werkzeug „curl".

Mit curl als Client können wir nun ganz einfach die Kommunikation zu einem beliebigen Server öffnen und Ressourcen anfordern bzw. über die bekannten HTTP-Methoden manipulieren. Die Ergebnisse der Anfragen werten wir für diese Tests dann manuell aus, d. h. wir wählen die Ressource (einen URI) und die HTTP-Methode (das Verb) für die nächste Anfrage aus. Kasten 3 zeigt die Verwendung von curl, um die Anfragen an einen lokal installierten Web-Server zu schicken, der auf dem Standard-Port horcht und die Bestellverwaltung so betreibt, wie in die-

curl zum Befüllen (POST-Anfrage) – Anwendungsfall „Bestellung aufgeben"

```
curl -d "Die erste Bestellung" http://localhost/customer/12/orders
```

curl zum Aktualisieren einer Bestellung (PUT-Anfrage) – Anwendungsfall "Bestellung ändern"

```
curl -T bestellung1.xml http://localhost/order/1
```

curl zum Anzeigen der Bestellungen eines Kunden (GET-Anfrage) – Anwendungsfall „Eigene Bestellungen anzeigen"

```
curl http://localhost/customer/0815/orders
```

curl zum Ansehen einer Bestellung (GET 120-Anfrage) – Anwendungsfall „Bestellung ansehen/laden"

```
curl http://localhost/order/order/1
```

Kasten 3: Einsatz von curl als Universal-Client

REST unterscheidet sich erheblich von klassischen RPC-Ansätzen. Das am stärksten hervorstechende Unterscheidungsmerkmal ist, dass beim REST-Ansatz keinerlei Versuch unternommen wird, die Tatsache zu verbergen, dass es sich um ein Netz-Kommunikationsmodell handelt – die von den RPC-Ansätzen angestrebte Ortstransparenz ist bewusst kein Designziel.

In einem REST-Modell erfolgt die Kommunikation typischerweise – wenn architekturkonform vorgegangen wird – zustandslos und grob-granular. Im Gegensatz dazu wird bei RPC, insbesondere bei Ansätzen mit verteilten Objekten, zustandsbehaftet entwickelt; die Interaktionen sind eher fein-granular (obwohl letzteres in der Literatur mehrheitlich als zu vermeidendes Negativszenario beschrieben wird).

Schließlich führt das Prinzip der generischen, uniformen Schnittstelle dazu, dass im Normalfall kein Code aus einer anwendungsspezifischen Schnittstellenbeschreibung generiert wird. Eine solche Beschreibung liegt im Gegenteil typischerweise gar nicht in maschinenlesbarer Form vor. Erst vor Kurzem wurde beim W3C eine lose Interessensgruppe gebildet, die Sinn und Unsinn von Beschreibungssprachen für die Schnittstellen REST-konformer Anwendungen diskutiert, bislang wurde allerdings noch keinen Konsens erzielt hat.

Kasten 4: *REST vs. RPC*

sem Artikel beschrieben. Die Implementierung unserer Server-Logik kann in jeder Programmiersprache erfolgen, die zur Erstellung von Web-Anwendungen geeignet ist – z. B. mit *Servlets* in Java, über CGI oder entsprechende Module mit einer Sprache wie Perl oder Ruby oder auch in einer beliebigen Kombination daraus.

Vor- und Nachteile von REST/HTTP

Vielfach wird davon ausgegangen, dass HTTP nur für die Online-Kommunikation geeignet und viel zu einfach ist, um für eine komplexe Anwendungsintegration genutzt zu werden. Zum einen sind es jedoch oft die einfachen Dinge, die für ein konkretes Problem viel besser geeignet sind, zum anderen stellt sich häufig erst bei der konkreten Anwendung heraus, dass das, was einfach scheint, in Wirklichkeit auf sehr fundierten und gar nicht trivialen Grundlagen basiert.

Neben der Einfachheit spricht für HTTP seine Reife und universelle Verbreitung. Dass HTTP funktioniert und interoperabel ist, kann nicht ernsthaft in Frage gestellt werden. Auch die Skalierbarkeit in Internet-Dimensionen ist belegt. Mit einer inkompatiblen Änderung des Protokolls ist aufgrund der extrem hohen Verbreitung auf absehbare Zeit nicht zu rechnen.

HTTP kann aus praktisch jeder Programmiersprache und -umgebung heraus verwendet werden, auch ohne auf besondere Toolkits zurückgreifen zu müssen. Die Verfügbarkeit unzähliger Werkzeuge – z. B. Kommandozeilen-Clients wie „curl" oder „wget", Werkzeuge zur Protokollanalyse, Proxy- und Firewall-Lösungen – spricht ebenfalls dafür. Auch die Integration mit der universellen Benutzungsschnittstelle Browser – zum Beispiel über die Möglichkeit, für ein und dieselbe Ressource Browser-Clients eine HTML-

Zunächst sollte zwischen Web-Services der ersten und zweiten Generation unterschieden werden – die erste Generation unterscheidet sich architektonisch kaum von vorhergehenden RPC-Ansätzen, sodass die Argumente übertragbar sind. Im Gegensatz dazu sind einige der Unterschiede zu RPC-Ansätzen auch bei moderneren Web-Service-Ansätzen wieder zu finden.

Sowohl bei Web-Services als auch bei REST-basierten Anwendungen wird in der Regel XML als Datenrepräsentationssprache verwendet. Beide Technologieansätze können zur Realisierung von Anwendungs-zu-Anwendungs-Kommunikation eingesetzt zu werden. Es ist durchaus möglich, auch in einer REST-konformen Anwendung SOAP einzusetzen, um Nachrichten mit *Header*-Elementen für die Abbildung nicht-funktionaler Charakteristika anzureichern.

Dennoch gibt es eine Reihe von Unterscheidungsmerkmalen: Ein wesentliches Entwurfsziel bei der Web-Services-Architektur ist die Transportunabhängigkeit. Web-Services sollen unabhängig vom eingesetzten Transportprotokoll sein. HTTP ist dabei neben z.B. SMTP/POP, JMS, IIOP, TCP und anderen nur eines von vielen Transportprotokollen. Anhänger der REST-Fraktion verwahren sich zu Recht gegen die Einordnung von HTTP als Transportprotokoll und legen Wert darauf, dass es sich um ein Applikationsprotokoll handelt, das auf einer anderen logischen Ebene liegt als z.B. TCP. Natürlich ist es möglich, HTTP als Transportprotokoll zu verwenden – wenn man bereit ist, sämtliche Vorteile des Web aufzugeben.

Das zentrale Element bei REST sind Ressourcen. Zwar erfolgt die Kommunikation mit diesen Ressourcen auch bei REST nachrichtenorientiert; in einem nachrichten- oder RPC-orientierten SOA-Szenario haben Ressourcen und URIs jedoch bestenfalls untergeordnete Bedeutung.

Ein weiterer Unterschied liegt in der Notwendigkeit zur Schnittstellenbeschreibung. REST definiert eine Schnittstelle, und zwar genau eine: die uniforme, d.h. allen Ressourcen gemeinsame.

Web-Service-Experten erkennen in der Regel den Wert der REST-Prinzipien an. So sind in die aktuellen Spezifikation zu SOAP und WSDL Elemente eingegangen, die auf dieser Erkenntnis beruhen (vgl. [W3C03], [W3C06]).

Für viele nicht-funktionale Anforderungen existieren im Umfeld der WS-*-Spezifikationen Lösungen, so z.B. für zuverlässige Nachrichtenübertragung (mit *WS-ReliableMessaging*), Sicherheit (*WS-Security*) oder für die Spezifikation von Eigenschaften (*WS-Policy*). Vergleichbare Standards existieren im REST-Umfeld nur teilweise; REST-Kritiker sehen dies als Schwäche, REST-Anhänger verweisen auf Standard-Lösungmuster oder sehen die Notwendigkeit gar nicht erst.

Kasten 5: *REST vs. Web-Services*

Repräsentation und Anwendungs-Clients eine XML-Repräsentation zur Verfügung zu stellen – ist ein positiver Faktor.

Eine REST-basierte HTTP-Anwendung verhält sich so, wie es sich für ein gewissenhaftes Mitglied einer Gemeinde – in diesem Fall der Web-Gemeinde – geziemt: Sie hält sich an die Regeln und entspricht den Erwartungen, die an sie gestellt werden. Als ein Beispiel sei die Absicherung von Ressourcen genannt, die von allen verbreiteten Web-Servern unterstützt wird. Hier lässt sich allgemein oder für autorisierte Benutzer auf Basis von Ressourcen und HTTP-Methoden sehr fein-granular festlegen, welche Methode auf welche Ressource angewandt werden kann. Mächtige Module wie „mod_rewrite" oder „mod_proxy" beim Apache-Server ermöglichen es, URIs dynamisch umzuschreiben, um Versionswechsel zu unterstützen, transparent für den Client eine Lastverteilung zwischen mehreren Servern vorzunehmen (und zwar ressourcen- und/oder methodenabhängig), häufig benötigte Inhalte statisch zwischenzuspeichern, temporäre oder dauerhafte Umleitungen einzurichten usw. – die Liste lässt sich noch lange fortsetzen. Kurz gefasst gilt, dass eine REST-Anwendung von allen Mechanismen, die das Web erfolgreich machen und sich als Reaktion auf Probleme im Laufe der Zeit neu etabliert haben, profitieren kann.

Eines der größten Probleme von HTTP ist die fehlende Unterstützung asynchroner Kommunikation. Hier muss entweder auf Anwendungsmuster oder nicht standardisierte Lösungen zurückgegriffen werden. Verschlüsselung und Authentifizierung erfolgen typischerweise auf Transport- und nicht auf Nachrichtenebene; allerdings ist eine Integration mit XML-*Security*-Mechanismen möglich. Die überaus nützliche GET-Methode ist implementierungsabhängig häufig mit einer Längenbegrenzung und immer mit „sichtbaren" Parametern versehen. Wenig verbreitete HTTP-Methoden – wie PUT und DELETE – sind oft in der Server- bzw. Firewall-Konfiguration gesperrt.

Fazit

Wir hoffen, wir konnten die Unterschiede zwischen der Entwurfsmetapher von REST-Anwendungen und der eher klassisch-objektorientierten oder komponentenbasierten Anwendung ein wenig verdeutlichen. Ein Ziel, das wir mit diesem Artikel verfolgen, ist es, einen gewissen Respekt für das oft unterschätzte Web und vor allem die ihm zu Grunde liegenden Architekturprinzipien zu erzeugen. Daneben gibt es jedoch auch diverse praxisrelevante Aspekte beim Einsatz von HTTP, dem REST-Protokoll per se.

Es in jedem Fall sinnvoll, sich mit der Architektur des Web auseinanderzusetzen. Ein Unternehmen wie Amazon.com, das sein API sowohl über Web-Services als auch über REST anbietet, wickelt 80% der Anfragen über die REST-Variante ab (**Kasten 4 und Kasten 5** geben hier noch ergänzende Hinweise). Wenn Sie Services nicht nur unternehmensintern, sondern -übergreifend anbieten wollen, sollten Sie REST als Realisierungsalternative gegenüber SOAP-basierten Web-Services in jedem Fall berücksichtigen. ■

Literatur & Links

[Apa] Apache, HTTP Server Project, siehe: httpd.apache.org/

[Fie00] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Doktorarbeit, Univ. of California, Irvine, 2000, siehe: www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[Int] International Resource Identifiers (IRIs); http://ietfreport.isoc.org/idref/rfc3987/

[Net99] Network Working Group, Hypertext Transfer Protocol – HTTP/1.1, 1999, siehe: ftp://ftp.isi.edu/in-notes/rfc2616.txt

[W3C01] W3C, URIs, URLs, and URNs: Clarifications and Recommendations 1.0, 2001, siehe: www.w3.org/TR/uri-clarification/

[W3C03] W3C, SOAP Version Part 2: Adjuncts, 2003, siehe: www.w3.org/TR/soap12-part2/#WebMethodFeature

[W3C06] Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts, 2006, siehe: www.w3.org/TR/wsdl20-adjuncts/#http-binding