5 Verben

Eine Anzahl von Operationen, die für alle Ressourcen gleichermaßen gültig sind – dies ist die Kernidee der »uniformen« (gleichförmigen) Schnittstelle. Formal legt Fielding für REST fest, dass es eine definierte, begrenzte Menge von Operationen geben muss, nicht aber, um welche genau es sich dabei handelt. Die konkrete Ausprägung von REST in HTTP dagegen definiert eine konkrete Liste von Verben (auch *Methoden* oder *Operationen genannt)*, die von Ressourcen unterstützt werden sollten. Für die Entwicklung von REST/HTTP-Anwendungen spielen diese daher eine zentrale Rolle.

5.1 Standardverben von HTTP 1.1

In der aktuellen Version von HTTP werden insgesamt acht Verben definiert: GET, HEAD, PUT, POST, DELETE, OPTIONS, TRACE und CONNECT. Betrachten wir zunächst die ersten sechs davon genauer.

5.1.1 GET

Die grundlegende und wichtigste Operation ist GET. Gemäß Spezifikation dient GET dazu, die Informationen, die durch die URI identifiziert werden, in Form einer *Entity* (d. h. einer Repräsentation) abzuholen. Die Operation ist als sicher *(safe)* definiert, ebenso wie das eng verwandte HEAD. Außerdem sind GET und HEAD *idempotent* (doch dazu später mehr).

Oft wird behauptet, GET dürfe keine *Seiteneffekte* erzeugen. Das ist – wie viele andere Aussagen über REST, die man häufig hört – weder ganz richtig noch ganz falsch. Ein Seiteneffekt tritt auf, wenn sich auf der Serverseite irgendein Zustand ändert, und dazu zählt rein formal auch schon das Erzeugen eines Eintrags in einer Logdatei. Ein solcher Effekt ist bei einem GET nicht nur üblich, sondern auch durchaus erlaubt. Entscheidend ist, dass der Client keine Zustandsänderung angefordert hat. Der Entwickler des Servercodes oder der Administrator in unserem Beispiel hat sich entschieden, GET-Requests zu protokollieren – für den Client ist das irrelevant. Es wäre absurd, wenn der Betreiber des Servers

seinen Nutzern eine Rechnung für den durch Logeinträge in Anspruch genommenen Plattenplatz schicken würde: »Sicher« bedeutet, dass der Nutzer mit dem Aufruf von GET und dem damit verbundenen Lesen von Daten keine Verpflichtung eingeht.

An dieser Stelle lässt sich gut illustrieren, warum man als Architekt einer Webanwendung die Semantik, die in der Spezifikation für die einzelnen HTTP-Verben definiert wird, unbedingt kennen sollte. Betrachten wir als Beispiel eine Webanwendung, die Fotos verwaltet und das Löschen eines Fotos über ein HTTP GET auf eine URI der Form http://example.com/pictures/operation=delete&id=1234 ermöglicht. Um das Löschen aus der HTML-Oberfläche zu ermöglichen, wird ein Link (<a href=... />) in die Seite aufgenommen: Das Verfolgen einer Verknüpfung führt nun zum Löschen einer Ressource. Ein Benutzer, der den Link benutzt, rechnet möglicherweise damit, weil er den Text lesen kann. Eine Suchmaschine jedoch geht davon aus, dass sie Links gefahrlos folgen darf – schließlich ist dies gemäß Spezifikation sicher. Traurig, wenn der hilfreiche Google-Crawler die Ressourcen löscht, über deren Indexierung die Betreiber der Foto-Community glücklich gewesen wären ...[1] Dieses Muster werden wir noch häufiger sehen: Die HTTP-Spezifikation definiert ein Verhalten, das Clients, Server und Intermediaries (also z. B. Proxy- oder Gateway-Knoten) im Web erwarten. Ob Sie sich an diese Erwartungen halten oder nicht, ist letztlich Ihre Entscheidung – ein Verstoß gegen die Spielregeln kann jedoch dazu führen, dass Sie Vorteile des Web nicht nutzen können oder dass sogar direkte Nachteile entstehen.

GET ist das Arbeitspferd des World Wide Web. Auch wenn genaue Zahlen nicht zu ermitteln sind, kann man annehmen, dass mehr als 95% aller Interaktionen im WWW lesenden Charakter haben – unabhängig davon, ob sich dahinter eine einfache Datei, ein Zugriff auf ein Content-Management-System (CMS) oder eine komplexe Anwendung verbirgt. Einer Verknüpfung zu folgen bedeutet, über HTTP ein GET an den Server zu senden, in dessen Hoheitsbereich die URI liegt, die man auflösen möchte. Bei unternehmensinternen Anwendungen mag der Anteil der Leseoperationen vielleicht geringer sein. Aber ob es nun 95, 85 oder 80% sind: Auf jeden Fall werden Informationen zu einem erheblich größeren Teil gelesen, als sie verändert, gelöscht oder um neue Informationen ergänzt werden.

Konsequenterweise ist GET die Operation, für die das gesamte Web optimiert ist. Ein Beispiel dafür ist das sogenannte bedingte GET (Conditional GET). Ein Client kann dem Server bei einem GET-Request über einem Header mitteilen, in welchen Fällen er eine Darstellung der Ressource überhaupt haben möchte. Mit dem If-Modified-Since-Header zum Beispiel teilt der Aufrufer mit, dass er

1. Man mag hier einwenden, dass ein solcher Link in der Regel nur für authentifizierte Benutzer und damit nicht für Google & Co. sichtbar ist. Aber selbst dann können Werkzeuge wie der Google Web Accelerator oder Browser-Plug-ins, die im Kontext der Benutzersitzung laufen, den gleichen Effekt haben (siehe auch [Fried2005]).

nur dann an einer Repräsentation interessiert ist, wenn diese sich seit einem definierten Datum geändert hat (in aller Regel ist dies das Datum, zu dem er sie zum letzten Mal abgeholt hat). Hat sie sich seitdem nicht geändert, antwortet der Server mit dem Ergebniscode 304 »Not Modified«. Allein über diese eine Optimierung lassen sich überflüssige Datentransfers in gewaltigem Umfang vermeiden; ein Polling – d. h. eine periodisch wiederholte Abfrage eines Status – wird auf einmal zu einer durchaus tolerablen Methode, Informationen über Änderungen zu verteilen. Mit dem Thema Caching und bedingten Anfragen beschäftigen wir uns im Detail in Kapitel 10.

Keine Operation, keine Methode, ist so universell verbreitet, so allgegenwärtig wie GET. Wenn Sie in der Fernsehwerbung, auf einem Plakat oder in Ihrer Tageszeitung eine URL lesen, wissen Sie (und auch Normalbürger, die ein Buch wie dieses sicher nie lesen würden), was Sie damit tun können: sie in die Adresszeile Ihres Webbrowsers einfügen und Enter drücken (und den Browser damit zum Absenden eines HTTP GET veranlassen). Auch wenn dies so nicht in der Spezifikation steht, gehört die Unterstützung von GET für ausnahmslos jede URI eigentlich zum guten Ton. Sie sollten sich angewöhnen, bei GET für jede Ihrer Ressourcen immer ein sinnvolles Ergebnis zurückzuliefern, und sei es auch nur eine aussagekräftige Fehlermeldung.[2]

5.1.2 HEAD

HEAD ist GET sehr ähnlich, mit einem entscheidenden Unterschied: Es wird keine Repräsentation der Ressource, auf die HEAD angewandt wird, zurückgeliefert, sondern nur die Metadaten (d. h. vor allem die Header). Über HEAD kann sich ein Client also über die Metadaten informieren, ohne die eigentlichen Daten transferieren zu müssen. Dies ist z. B. sinnvoll, um die Existenz einer Ressource zu prüfen, um sicherzustellen, dass die Länge der Repräsentation im verarbeitbaren Rahmen liegt, oder um den Zeitpunkt der letzten Änderung herauszufinden. Die Spezifikation schreibt vor, dass beim HEAD exakt die gleichen Metadaten zurückgeliefert werden müssen wie bei einem GET auf die gleiche URI.

Für die Implementierung von HEAD haben Sie bei der Entwicklung Ihres Servers zwei Optionen: Entweder führen Sie die gleiche Logik aus wie bei der Behandlung eines GET, verwerfen jedoch die bereits ermittelten Daten, anstatt Sie an den Client zu senden. Da das in aller Regel ineffizient ist, sollten Sie besser die zweite Option wählen und nur die für die Berechnung der Metadaten notwendige Verarbeitung durchführen, d. h. GET und HEAD unterschiedlich behan-

2. Um die Transparenz Ihres Systems zu maximieren, empfiehlt sich eine Plaintext- und/oder eine HTML-Repräsentation zu unterstützen – Sie erleichtern damit sich und anderen die Fehlersuche und machen Ihre Ressource noch ein Stück weit mehr zum Teil des WWW. Mehr dazu finden Sie in Kapitel 7.

deln. Diese Sonderbehandlung einzelner Methoden werden wir noch häufiger thematisieren: Die Kenntnis von HTTP-Protokoll und REST-Prinzipien ermöglicht es Ihnen, bestimmte Anfragen in Ihren Anwendungen so zu behandeln, dass der Ressourcenverbrauch deutlich sinkt und Performance und Durchsatz steigen.

5.1.3 PUT

Mit PUT wird eine bestehende Ressource aktualisiert oder, falls sie noch nicht vorhanden ist, erzeugt. Ein PUT wirkt sich direkt auf die Ressource aus, deren URI Ziel des Requests ist. Damit ist PUT die inverse Operation zu GET. Zur Übermittlung der Informationen über den gewünschten Zustand der Ressource verwendet der Client den sogenannten *Entity Body* der HTTP-Nachricht und informiert den Server im Content-Type-Header über das Format, in dem er die Informationen übermittelt. Ein HTTP-Client erwartet dabei, dass die Ressource *sinngemäß* mit den Informationen angelegt oder aktualisiert wird, die er übermittelt hat. Das klingt sehr weich; gemeint ist damit, dass der Server nicht alle Daten berücksichtigen muss und einige davon ändern, ignorieren oder durch neue ergänzen darf.

Eine besondere Eigenschaft, die PUT mit GET, HEAD und DELETE gemeinsam hat, ist die Idempotenz. Das ist eigentlich ein mathematischer Begriff, der eine Klasse von Funktionen definiert, die das gleiche Resultat liefern, wenn man sie auf sich selbst anwendet (d. h., bei denen f(x) = f(f(x)) ist). In der Literatur zu verteilten Systemen wird diese Definition ein wenig aufgeweicht: Eine Operation ist dann idempotent, wenn ein mehrmaliges Aufrufen die gleichen Seiteneffekte bewirkt wie ein einmaliges.

Was zunächst sehr theoretisch klingt, ist ein wesentliches Element einer stabilen Gesamtarchitektur: Nehmen wir an, Ihr Client sendet eine Anfrage, bekommt aber keine Antwort. Ist nun die Anfrage nicht angekommen, d. h., hat auf dem Server vielleicht gar keine Verarbeitung stattgefunden? Oder ist die Anfrage korrekt verarbeitet worden und nur die Antwort verloren gegangen? Will man diese beiden Fälle unterscheiden, ist eine aufwendige, wechselseitige Bestätigungskette notwendig. HTTP setzt mit PUT und den anderen idempotenten Methoden stattdessen darauf, dass es der Client im Zweifelsfall noch einmal versucht, und liefert die dafür notwendige Garantie: Weil die Spezifikation definiert, dass diese Operationen idempotent sind, darf der Client bei diesen (und nur diesen) so verfahren. Ähnlich wie bei GET ist es auch hier Aufgabe des Serverdesigners (also wahrscheinlich Ihre!), die Methoden korrekt – also hier: spezifikationskonform – zu implementieren und nicht gegen die Erwartungen des Clients zu verstoßen.

5.1.4 POST

Die Methode POST hat zwei Bedeutungen, eine enge und eine erweiterte. Im engeren Sinne bedeutet POST das Anlegen einer neuen Ressource unter einer vom Server bestimmten URI. Im weiteren Sinne wird POST für alle die Zwecke eingesetzt, in denen keine der anderen Methoden passt, d. h. immer dann, wenn eine beliebige Verarbeitung angestoßen werden soll.

Sowohl POST als auch PUT können damit für das Anlegen neuer Ressourcen verwendet werden. Im Gegensatz zu einem PUT gibt der Client bei einem POST jedoch nicht die URI der Ressource an, die er neu anlegen möchte, sondern die URI der für das Anlegen zuständigen Ressource. Häufig handelt es sich dabei um *Listenressourcen*, denen ein neues Element zugeordnet werden soll (siehe Abschnitt 4.2.3). Das Neuanlegen von Ressourcen über POST ist ein Standardmuster, für das auch ein passender HTTP-Statuscode existiert: »201 Created«. Mit dieser Antwort informiert der Server den Client darüber, dass die Ressource korrekt angelegt werden konnte. Die URI wird damit nicht vom Client, sondern vom Server bestimmt und über einen Location-Header dem Client bekannt gegeben.

Die zweite Bedeutung von POST ist sehr viel allgemeiner: Im Prinzip kann POST benutzt – mancher würde sagen: missbraucht – werden, um beliebige Funktionalitäten anzustoßen. Welche Funktionalität genau das ist, wird dabei in den übermittelten Daten codiert. Es wird damit zum letzten Ausweg – alles, was man nicht anders abbilden kann, wird auf POST abgebildet. Dies entspricht nicht dem Sinne des Erfinders[3], wird in der Praxis aber dennoch häufig benötigt.

Ebenso wie durch GET lassen sich auch durch POST beliebige Operationen »tunneln«. Im Falle von POST ist dies jedoch nicht ganz so dramatisch, da die HTTP-Spezifikation über die Semantik von POST ohnehin keine Garantien liefert: Weder kann eine Antwort gecacht werden, noch ist die Operation idempotent oder sicher. Bildet man Aktionen auf POST ab, die besser mithilfe einer anderen HTTP-Methode aufgerufen worden wären, verstößt man nicht explizit gegen eine Vorschrift und handelt sich die damit verbundenen Nachteile ein, sondern man nutzt stattdessen die Vorteile der anderen Methoden nicht. Man könnte die Fehlnutzung von GET mit Verlust, den Missbrauch von POST mit entgangenem Gewinn vergleichen.

5.1.5 DELETE

Wie zu erwarten ist DELETE für das Löschen der Ressource zuständig, deren URI im Request angegeben wird. Die Methode ist idempotent, weil der Effekt eines mehrmaligen Löschens der gleiche wie der eines einmaligen Löschens ist.

---

3. Im wahrsten Sinne des Wortes, siehe [Fielding2000].

Da Ressourcen nur eine äußere Sicht auf eine Anwendung darstellen, kann man sich das Löschen per DELETE als logisches Löschen vorstellen: Es ist durchaus denkbar und üblich, in der internen Datenrepräsentation – der Persistenzschicht – nur ein entsprechendes Attribut zu setzen (z. B. »storniert«). Für den Client ist dies unerheblich; das von außen sichtbare Verhalten zeigt, dass die Ressource nicht mehr existiert.

5.1.6 OPTIONS

Die OPTIONS-Methode hat eine Sonderrolle: Sie liefert Metadaten über eine Ressource, unter anderem im Allow-Header über die Methoden, die eine Ressource unterstützt. OPTIONS ist idempotent und sicher; wenn nicht explizite Cache-Header gesetzt sind, darf das Resultat vom Client nicht gecacht werden. Die Implementierung eines Servers sollte OPTIONS unterstützen, ein Client sollte sich jedoch nicht darauf verlassen.

5.1.7 TRACE und CONNECT

Der Vollständigkeit halber möchten wir die Methoden TRACE und CONNECT noch erwähnen: TRACE dient zur Diagnose von HTTP-Verbindungen, in denen möglicherweise eine Reihe von Intermediaries Nachrichten filtern und verändern. Gemäß Spezifikation sendet der Server eine Kopie der Nachricht, ergänzt um eine Reihe von Via-Headern. In der Praxis wird sie kaum unterstützt und daher praktisch nie verwendet. CONNECT dient zur Initiierung einer Ende-zu-Ende-Verbindung bei der Verwendung von SSL durch einen Proxy.

5.2 HTTP-Verben in der Praxis

In der folgenden Tabelle finden Sie die HTTP-Methoden noch einmal im Überblick, zusammen mit einer Charakterisierung anhand der Eigenschaften, die durch die Spezifikation vorgegeben werden. Sicher bezeichnet dabei die Abwesenheit unerwünschter Seiteneffekte, Idempotenz die Möglichkeit zur Wiederholung im Zweifelsfall. Die nächsten beiden Spalten geben an, ob die Ressource, auf die die Methode »wirkt«, durch die URI identifiziert wird, und ob das Ergebnis gecacht bzw. aus einem Cache gelesen werden kann. (Das »O« bei der OPTIONS-Methode bedeutet, dass zwar gecacht werden kann, dies jedoch nicht die Voreinstellung ist und explizit angezeigt werden muss.) Die Spalte »sichtbare Semantik« schließlich zeigt an, ob die Infrastruktur Kenntnis von der Semantik der Methode haben kann.

An der Tabelle können Sie erkennen, dass es nahezu keine Garantien bei »POST« gibt – genau aus diesem Grund kann sie als Notlösung für all die Fälle dienen, in denen man bewusst gegen die REST-Restriktionen verstoßen möchte.

Auf der anderen Seite ist sie damit auch die am wenigsten nützliche Methode, die am wenigstens Transparenz über das durch sie angestoßene Verhalten liefert.

5.3 Tricks für PUT und DELETE

5.3.1 HTML-Formulare

In diesem Buch liegt unser Hauptaugenmerk auf der Kommunikation zwischen Anwendungssystemen. Dennoch werden Sie häufig mit der Anforderung konfrontiert, Ressourcen sowohl über den Browser als auch über ein API verfügbar zu machen. Unglücklicherweise gibt es dabei ein Problem: HTML-Formulare unterstützen nur GET und POST, nicht aber PUT und DELETE.

HTML ist ein ebenso wichtiger Bestandteil des WWW wie HTTP und URIs. Es ist deswegen nicht verwunderlich, dass HTML in vielerlei Hinsicht perfekt zu REST passt. Es erstaunt daher umso mehr, dass ausgerechnet HTML nicht alle HTTP-Methoden unterstützt, sondern künstlich auf GET und POST begrenzt. Zukünftige HTML-Versionen versprechen Abhilfe, die Verfügbarkeit liegt allerdings zum Zeitpunkt, zu dem wir diese Zeilen schreiben, noch in ferner Zukunft [formHTTP, formJSON].

Wenn Sie die Semantik Ihrer Ressourcen in der Praxis korrekt auf PUT und DELETE abbilden wollen, müssen Sie deshalb eine Lösung finden, mit der Sie diese Einschränkung umgehen können. Dazu stehen Ihnen zwei Varianten zur Verfügung: die Nutzung eines versteckten Formularfelds oder der Einsatz von Ajax.

Methode	sicher	idempotent	identifizierbare Ressource	Cache-fähig	sichtbare Semantik
GET	X	X	X	X	X
HEAD	X	X	X	X	X
PUT		X	X		X
POST					
OPTIONS	X	X		O	X
DELETE		X	X		X

Tab. 5–1 *HTTP-Methoden und ihre Eigenschaften (nach [Fielding2004])*

Versteckte Formularfelder

Sehen wir uns die Struktur eines einfachen HTML-Formulars kurz an:

```
<form action='/items/1' method=POST>
  <input type='text' name='name'  value='old value'/>
  <input type='submit' value='Update item'/>
</form>
```

Der Wert des Attributes »method« bestimmt, ob die Inhalte des Formulars per POST an den Server übermittelt werden oder ob daraus eine URI konstruiert und darauf ein GET durchgeführt wird. In unserem Fall führt ein Ändern des Wertes und ein Klick des Buttons »Update item« zu folgendem Request:

```
POST /items/1 HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_6; en-us)
AppleWebKit/525.27.1 (KHTML, like Gecko) Version/3.2.1 Safari/525.27.1
Content-Type: application/x-www-form-urlencoded
Referer: http://localhost:4567/items/1
Accept: text/xml,application/xml,text/html;q=0.9,text/plain;q=0.8,*/*;q=0.5

Content-Length: 26
Connection: keep-alive
Host: localhost:4567

name=new+value
```

Es gibt leider keine Möglichkeit, über den Formularmechanismus dem Browser beizubringen, ein PUT zu verwenden. Aber wir können ein weiteres Feld in unser Formular aufnehmen:

```
<input type='hidden' name='_method' value='PUT'>
```

Natürlich übermittelt der Browser die Daten immer noch in Form eines POST-Requests, die Logik auf der Serverseite – idealerweise gekapselt in einem Framework – kann aber dafür sorgen, dass eine solche Anfrage an die gleiche Funktion weitergeleitet wird wie ein »echter« PUT-Request. Analog dazu lässt sich mit DELETE verfahren. Für den Entwickler des Servers bedeutet das, dass er seine Logik so implementieren kann, als wäre die HTML-Welt in Ordnung. Dieser Workaround wird in verschiedenen Frameworks für die Entwicklung von Webanwendungen unterstützt, unter anderem auch von Ruby on Rails.

Ajax

Über Ajax (Asynchronous JavaScript and XML [Ajax]), das in der Praxis weder asynchron sein noch etwas mit XML zu tun haben muss, können Sie aus JavaScript HTTP-Anfragen an den Server senden. Dazu verwendet man ein JavaScript-API, das XmlHttpRequest-Objekt. Dabei gibt es (zumindest in aktuellen Browsern) keine Begrenzung auf GET und POST, in jedem Fall werden PUT und DELETE überall unterstützt.

Modifizieren wir dazu unser Beispiel: Anstelle des versteckten Formularparameters definieren wir eine JavaScript-Funktion[4], die wir als Eventhandler für die Freigabe des Formulars registrieren:

4. In diesem Beispiel verwenden wir die JavaScript-Bibliothek JQuery [JQuery], andere Bibliotheken wie Prototype [Prototype] oder die direkte Nutzung von JavaScript und XmlHttpRequest wären ebenso möglich.

```
<form id="item_form" action='/items/1' method='POST'>
  <input type='text'name='name'  value='old value'/>
  <input type='submit' value='Update item'/>
</form>
<script type="text/javascript">

  $(document).ready(function() {
      var form = $('#item_form');
      form.submit(function() {
          var action = form.attr("action");
          var serializedForm = form.serialize();
          $.ajax({
              type: "PUT",
              url: action,
              data: serializedForm
          });
          return false;
      });
  });
</script>
```

Für JavaScript-Unkundige eine kurze Erklärung: Das Formular unterscheidet sich kaum vom vorherigen, allerdings verzichten wir auf das versteckte Formularfeld. Neu hinzugekommen ist ein id-Attribut, das dem Formular einen Namen gibt. Diesen Namen verwenden wir weiter unten, um eine JavaScript-Funktion zu registrieren, die beim Klick auf »Update item« aufgerufen wird. In dieser senden wir mithilfe eines Ajax-Aufrufs die Daten aus dem Formular an die dort mit dem Attribut »action« spezifizierte URI.

Vor- und Nachteile

Beide Varianten, das »hidden field« und der Ajax-Aufruf, sind nicht optimal. Vorteil der ersten Variante ist, dass Sie auf den Einsatz von JavaScript nicht angewiesen sind. Der Hauptnachteil ist, dass die Infrastruktur nur einen HTTP-POST-Request »sieht« und nichts von der eigentlichen Absicht erfährt, da diese im Formularfeld versteckt ist. Bei der Ajax-Variante ist es genau andersherum: Sie benötigen nun zwingend JavaScript, dafür aber wird ein »echter« PUT-Request versandt.

Sie können beide Varianten kombinieren, sollten in der Praxis allerdings abwägen, ob sich die Mühe lohnt – in der Regel ist der Einsatz des versteckten Formularfelds das geringere Übel, es sei denn, Ihre Benutzeroberfläche ist auf JavaScript ohnehin angewiesen.

5.3.2 Firewalls und eingeschränkte Clients

Eine weitere Restriktion für die Verwendung von PUT und DELETE ergibt sich aus der Standardkonfiguration einiger Firewallprodukte und anderer Intermedi-

ary-Knoten, die zwar GET- und POST-Anfragen passieren lassen, PUT und DELETE (und auch weitere Methoden) jedoch blockieren. Diese Einschränkung stammt aus der Zeit, in der man PUT und DELETE auf statische Webseiten bezog, und mag zu diesem Zeitpunkt sinnvoll gewesen sein; heute ist ein POST in der Regel genauso sicherheitsrelevant.

Der *richtige* Weg ist, dafür zu sorgen, dass die Methoden in den Intermediaries, die Ihre Anfragen auf ihrem Weg zum endgültigen Zielserver passieren, freigeschaltet sind. Es gibt jedoch mindestens zwei Situationen, in denen dies nicht so einfach ist, wie es sich anhört: Zum einen sind Sie als Anbieter einer öffentlichen Schnittstellen gar nicht in der Position, in die Infrastruktur Ihrer Benutzer einzugreifen oder diesen über Gebühr Auflagen zu machen. Zum anderen kann es selbst bei unternehmensinternen Anwendungen an sozialen Aspekten scheitern, typisches Beispiel: Eine progressive Gruppe in einem Großunternehmen möchte einen externen Dienst nutzen, aber die zentrale IT-Administration, die schon fast aus Pflicht konservativ sein muss, lässt sich nicht so einfach überzeugen.

Für diese Situation gibt es einen ähnlichen Workaround wie für die Begrenzung bei HTTP-Formularen: Sie verwenden ein HTTP POST und »tunneln« das eigentliche Verb durch einen Nebenkanal – in diesem Fall durch einen HTTP-Header. Diese Lösung wurde vor allem dadurch bekannt, dass Google sie für die GData-APIs [GData] einsetzt. Für Fälle, in denen PUT- bzw. DELETE-Requests irgendwo auf dem Weg zwischen Client und Server ungewollt blockiert werden, wird stattdessen ein Header gesetzt[5]:

```
X-HTTP-Method-Override: PUT
```

Auch in diesem Fall wird die POST-Anfrage vom Server wie ein PUT interpretiert; analog ist dies auch mit DELETE möglich. Und ähnlich wie bei den HTML-Tricks sollten Sie auch diese Variante durchaus kritisch betrachten: Natürlich ist es ärgerlich, wenn ein Administrator eine Firewall aus Unkenntnis falsch konfiguriert hat. Aber wer sagt Ihnen, dass das der Grund ist? Vielleicht wollte jemand ganz explizit genau diese Art von Zugriff blockieren – warum haben Sie das Recht, diese Einschränkung durch ein Verstecken der eigentlichen Intention zu umgehen?

Auch hier sollten Sie daher gut überlegen, wie Sie vorgehen. Falls Sie ein Server-API entwickeln, schadet es nicht, den oben beschriebenen oder einen äquivalenten Header korrekt zu interpretieren – die Entwickler der Clients können dann selbst entscheiden, ob sie ihn nutzen oder nicht. Als Cliententwickler sollten Sie grundsätzlich zunächst versuchen, Ihre Partner in der Systemadministration von den Vorteilen einer expliziten Protokollnutzung zu überzeugen.

Anders ist die Situation, wenn Sie einen Zugriff auf Ihre Dienste auch Clients ermöglichen wollen, die PUT-und DELETE-Requests schlicht nicht erzeugen

5. Header, die mit dem Präfix »X-« beginnen, können Sie frei definieren; ansonsten müssen Sie den offiziellen und in [RFC3864] dokumentierten Weg gehen.

können. Das gilt z. B. für Java ME, die eingeschränkte Java-Umgebung für mobile Geräte – hier können Sie nur GET, HEAD und POST als Methoden verwenden.[6] Für diesen Fall müssen Sie einen Workaround wie den des zusätzlichen Headers vorsehen.

5.4 Definition eigener Methoden

Reichen Ihnen die von HTTP definierten und in den gängigen Werkzeugen und Plattformen unterstützten Verben nicht aus, können Sie eigene definieren. Das sollte allerdings die absolute Ausnahme sein – in aller Regel werden die wenigen Vorteile durch die Nachteile mehr als aufgewogen. Betrachten wir diese Vor- und Nachteile genauer.

Zunächst einmal kann es vorkommen, dass eine Operation von einer Vielzahl von Ressourcen unterstützt wird und überall die gleiche Semantik hat. Ein Beispiel wäre eine Operation CHECKOUT[7]: Sie ist potenziell für jede Ressource gültig und würde daher durchaus den REST-Prinzipien entsprechen. Gleichzeitig vermeiden Sie durch die Definition eines eigenen Verbs das »Überladen« einer anderen Operation. Die Absicht, die Sie mit Ihrer Operation verfolgen, wird damit auch auf der Protokollebene explizit. Sie könnten dadurch z. B. die Sicherheitsmechanismen des Apache-Webservers nutzen, um bestimmten Nutzern den Zugriff zu erlauben und ihn anderen zu verwehren.

Der große Nachteil ist jedoch, dass nur die Clients, die die von Ihnen neu definierte Operation kennen, mit Ihrer Anwendung zusammenarbeiten können. Sie haben damit einen der größten Vorteile des Einsatzes von RESTful HTTP zunichte gemacht, weil das Ökosystem von potenziellen Partnern, die mit den Ressourcen kommunizieren können, deutlich kleiner ist. Wie groß das Ökosystem ist, hängt von dem Geltungsbereich Ihrer Vereinbarung ab: Falls Sie für die komplette IT-Landschaft eines Großunternehmens verantwortlich sind, können Sie sich einen Sonderweg vielleicht eher erlauben – obwohl auch in diesem Fall die mangelnde Unterstützung durch Standardsoftware ein Problem sein kann.

5.4.1 WebDAV

WebDAV (»Web-byased Distributed Authoring and Versioning« [WebDAV]) ist ein Beispiel für ein Protokoll, das HTTP mit einem Satz eigener Verben und Header (und einer XML-basierten Syntax für die Darstellung von Metadaten) erweitert. Es dient zur gemeinsamen Bearbeitung von Dateien, die von einem WebDAV-

6. Dafür gibt es selbst bei sehr viel Fantasie keinen einzigen plausiblen Grund, aber das ändert leider nichts daran.
7. Im Sinne eines Versionskontrollsystems: Die Methode könnte eine Ressource für eine exklusive Bearbeitung durch einen Client sperren und in dieser Zeit anderen Clients ausschließlich das Lesen erlauben.

fähigen Server verwaltet werden, und wird zum Beispiel von Microsoft Exchange, dem Versionskontrollsystem Subversion und diversen Clients (z. B. dem Explorer unter Windows oder dem Finder auf Mac OS X) unterstützt.

Die folgende Tabelle gibt eine Übersicht über die wichtigsten WebDAV-Methoden[8]:

WebDAV-Methode	Bedeutung
PROPFIND	Liefert Eigenschaften (Properties) und deren Werte für eine Ressource zurück
PROPPATCH	Ändert oder entfernt Eigenschaften
MKCOL	Legt eine neue Collection-Ressource (analog zu einem Ordner/Verzeichnis) an
COPY	Kopiert eine Collection, Ressource oder Eigenschaft(en)
MOVE	Verschiebt eine Collection oder Eigenschaft(en)
LOCK	Sperrt eine Ressource
UNLOCK	Entsperrt eine Ressource
VERSION-CONTROL	Erzeugt eine Ressource unter Versionskontrolle
REPORT	Liefert Informationen über die Metadaten einer Ressource
CHECKOUT	Muss aufgerufen werden, bevor eine unter Versionskontrolle stehende Ressource modifiziert werden kann
CHECKIN	Erzeugt eine neue Version einer Ressource
UNCHECKOUT	Macht ein CHECKOUT rückgängig
UPDATE	Setzt eine Ressource auf eine andere Version
LABEL	Gibt einer Version einen Namen
MERGE	Konsolidiert zwei Versionen einer Ressource

Tab. 5–2 *WebDAV-Methoden*

Die Designer des WebDAV-Protokolls haben sich entschieden, den Erweiterungsmechanismus von HTTP zu nutzen. Entspricht das Ergebnis den REST-Prinzipien? Grundsätzlich ja, allerdings gibt es einen wesentlichen Kritikpunkt: Ressourcenversionen, die unter der Kontrolle eines WebDAV-konformen Servers stehen, sind weniger sichtbar, als sie es bei der Abbildung auf die Standard-HTTP-Methoden wären. So können weder die Metadaten noch eine andere als die aktuelle Version einer Ressource über ein GET abgefragt werden, wodurch die Möglichkeit zur Integration in das Standard-HTTP-Ökosystem deutlich eingeschränkt wird. Das klingt zunächst recht abstrakt, aber wenn Sie schon einmal Subversion benutzt haben, haben Sie sich vielleicht darüber geärgert, dass Sie niemandem einen Link auf eine bestimmte Version per Mail zusenden können.

8. Als Summe aus den Methoden, die im Basis-WebDAV-Standard [RFC4918] und in den Erweiterungen für Versionierung [RFC3253] definiert werden.

Bereits im letzten Abschnitt haben wir davon abgeraten, die Menge von Standardverben um eigene zu erweiteren. Wenn Sie sich aber dennoch dafür entscheiden, ist WebDAV ein gutes Vorbild. Bei der Diskussion von AtomPub in Kapitel 8 werden wir erfahren, wie ein Entwurf aussehen kann, der sich mit einer ähnlichen Thematik beschäftigt, aber auf eigene Verben verzichtet.

5.4.2 Partial Updates und PATCH

Die richtige HTTP-Methode für das Aktualisieren einer Ressource ist PUT. Diese Methode ist jedoch so definiert, dass der übertragene Inhalt eine vollständige Repräsentation des Ressourcenstatus sein soll – mit anderen Worten: Bei einem PUT wird der *gesamte* Inhalt übertragen.

Der Server kann zwar Teile davon ignorieren oder interpretieren, er kann aber nicht daraus, dass Sie einzelne Elemente weglassen, darauf schließen, dass Sie diese unverändert lassen möchten. Nehmen wir an, Sie erhalten als Ergebnis eines HTTP GET folgendes XML-Dokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
   <last>Doe</last>
   <first>John</first>
</person>
```

Nun senden Sie eine Aktualisierung via PUT:

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
   <first>Jack</first>
</person>
```

Was ist Ihre Erwartungshaltung? Haben Sie nun nur den Vornamen geändert oder auch den Nachnamen auf »Leer« gesetzt? In diesem Fall ist die erste Option die wahrscheinlichere, verlassen können wir uns darauf jedoch nicht. Alternativ könnten wir ein eigenes Format erfinden, in dem wir die gewünschten Änderungen beschreiben:

```
<?xml version="1.0" encoding="UTF-8"?>
<patch>
   <change xpath='/first'>Jack</change>
</patch>
```

Aber wenn dieses Fragment per PUT versandt wird, ist die vordefinierte Semantik »ersetze den Zustand der Ressourcen mit dem in dieser Repräsentation beschriebenen«. Der Server müsste sich also je nach Content-Type anders verhalten – womit die Methode nicht mehr klar aussagt, welche Semantik vorliegt. Es gibt noch ein weiteres Argument gegen PUT für eine teilweise Aktualisierung *(Partial Update)*. Dazu sehen wir uns noch ein leicht modifiziertes Beispiel an:

```
<?xml version="1.0" encoding="UTF-8"?>
<patch>
   <add path='/first'><element name="middle">Q.</element></add>
</patch>
```

In unserem Format soll damit ausgedrückt werden, dass ein neues Element hinzugefügt werden soll. Aber was passiert, wenn wir vom Server keine Antwort erhalten, uns auf die Idempotenz-Garantie von PUT verlassen und es einfach noch einmal versuchen? Wie kann der Server entscheiden, ob wir wirklich ein weiteres Element hinzufügen wollten oder es nach einem Fehlschlag noch einmal versuchen?

Die einzig sinnvolle Lösung für ein teilweises Update im Rahmen der vorgegebenen HTTP-Verben ist ein POST, da es hier keine Semantik gibt, gegen die wir verstoßen könnten. An dieser Stelle setzt ein neu standardisiertes Verb an: PATCH [RFC5789]. Diese weder »sichere« noch idempotente Methode würde definiert als Aktion, bei der die übertragenen Daten ein zur Ressource passendes Format für den Ausdruck der Änderungen verwenden würden.

Solange das Verb noch nicht verfügbar ist, gibt es zwei denkbare Lösungen.

Die erste Lösung besteht darin, die Änderungen per POST zu übertragen. Ob Sie dabei ein generisches Format verwenden, das die Unterschiede ausdrückt, oder ein für Ihre Anwendung spezifisches, hängt vor allem davon ab, welches Format Sie für die vollständige Übertragung verwenden: Wenn Ihre Ressource eine rein textuelle Repräsentation hat (und auch darüber geändert werden kann), kommen Sie möglicherweise mit einem textbasierten Diff-Format aus, wie es das Standard-Unix-Werkzeug diff produziert. Für XML können Sie XSLT oder XQuery verwenden; für JSON finden Sie eine elegante Lösung unter [Sayre 2007].

Ein weiterer Lösungsansatz besteht darin, die Ressource in mehrere untergeordnete Ressourcen aufzuteilen, die wiederum einzeln aktualisiert werden können. Diese können sich untereinander überlappen; Sie müssen dann im Rahmen Ihrer Serverimplementierung dafür sorgen, dass sich die Änderungen auf alle indirekt betroffenen Ressourcen ebenfalls auswirken.

5.4.3 Multi-Request-Verarbeitung

Häufig müssen Sie zur Erledigung einer Aufgabe eine ganze Reihe von Ressourcen auf einmal bearbeiten. Die naive Lösung – die allerdings oft genug völlig ausreichend ist –, besteht darin, die einzelnen Requests nacheinander vom Client aus zum Server zu senden.

Es liegt auf der Hand, dass dieser Ansatz gerade bei sehr großen Datenmengen nicht immer praktikabel ist, denn zur Verarbeitungszeit für jeden einzelnen Request auf der Serverseite müssen Sie noch die Netzwerklatenz hinzurechnen. Ein einfacher – und der aus meiner Sicht beste – Weg ist es, für diesen Fall auf der

Serverseite eine eigene Ressource zu genau diesem Zweck anzulegen. Sie können damit die Durchführung ihres Verarbeitungslaufs als eine Neuanlage einer Ressource betrachten, über deren Status Sie sich während und auch noch nach der Verarbeitung per HTTP GET informieren können. Voraussetzung dafür ist, dass Sie die gewünschte Semantik sinnvoll auf eine solche Ressource abbilden können.

Betrachten wir als Beispiel eine Listenressource für Kunden unter /customers, die Sie nach dem in Abschnitt 4.2.3 beschriebenen Muster modelliert haben und der Sie demnach per POST neue Kunden hinzufügen können. Einzelne Kunden aktualisieren Sie über ein PUT. Falls eine zusätzliche Anforderung darin besteht, dass mehrere Kunden – sagen wir, einige Tausend – gleichzeitig aktualisiert werden sollen, ist das Senden einzelner Requests sehr ineffizient. In diesem Fall könnten Sie eine neue Ressource für den Zweck des Batch-Updates definieren (/customers/batchupdate) und mehrere Änderungen auf einmal per POST übermitteln:

```
POST /customers/batchupdate HTTP/1.1
Content-Type: application/vnd.mycompany.multi-customer-update+xml
Accept: text/plain
Content-Length: ...
Connection: keep-alive
Host: example.com

<?xml version="1.0" encoding="UTF-8"?>
<multi-update>
    <customer id='/customers/1234'>
      <name='XYZ' />
    </customer>
    <customer id='/customers/7362'>
      <name='ABC' />
    </customer>
    <customer id='/customers/1111'>
      <name='ZZZ' />
    </customer>
    <!-- ... -->
    <customer id='/customers/4321'>
      <name='KLF' />
    </customer>
</multi-update>
```

Sie haben damit eine sehr spezifisch auf Ihren Anwendungsfall zugeschnittene Lösung gewählt – und in der Regel ist dies der beste Weg.

Der Vollständigkeit halber möchten wir jedoch einen alternativen Ansatz nicht unerwähnt lassen, der versucht, das Problem generisch zu lösen. Dazu sendet der Client eine MIME-Multipart-Nachricht und verwendet den Content-Type multipart/mixed. Eine solche Nachricht besteht aus mehreren Teilen, von denen wiederum jeder einzelne einen eigenen Medientyp haben kann – in unserem Fall application/http:

```
Content-Type: multipart/mixed; boundary=msg

--msg
Content-Type: application/http;version=1.1
Content-Transfer-Encoding: binary
POST /customers HTTP/1.1
Host: example.com
Content-Type: application/ vnd.mycompany.customer+xml

<?xml version="1.0" encoding="UTF-8"?>
<customer>…</customer>

--msg
Content-Type: application/http;version=1.1
Content-Transfer-Encoding: binary

PUT /customers/7362 HTTP/1.1
Host: example.com
Content-Type: application/vnd.mycompany.customer+xml

<customer>
   <name='ABC' />
</customer>
--msg--
```

Die Nachricht besteht somit aus einer Reihe einzelner HTTP-Requests, die Sie in einem Block an den Server übermitteln. Ziel könnte auch in diesem Fall eine Ressource sein, die speziell für diesen Fall zur Verfügung gestellt wird.

Ähnlich wie bei der Verwendung von POST für ein Partial Update stellt sich auch hier die Frage, mit welchem Verb ein solcher Batchauftrag an den Server übermittelt werden sollte. Und auch hier lässt sich argumentieren, dass POST keine gute Wahl ist, sondern ein spezifisches Verb eingeführt werden sollte. Konsequenterweise gab es dafür auch einen entsprechenden Standardisierungsvorschlag, der das Verb BATCH für diesen Zweck vorsieht – er hat jedoch keine Akzeptanz gefunden.

5.5 LINK und UNLINK

Wir werden uns erst in Abschnitt 6.9 mit sogenannten Link-Headern beschäftigen, die Verknüpfungen zwischen beliebigen Ressourcen unabhängig vom Repräsentationsformat darstellen können. Dennoch möchten wir der Konsistenz wegen die im Moment noch in der Standardisierung befindlichen HTTP-Verben LINK und UNLINK zumindest erwähnen. Diese Verben können Sie verwenden, wenn Sie diese Links per HTTP setzen oder entfernen möchten. Diese Verben haben noch keine große Verbreitung, aber mit der zunehmenden Relevanz von Link-Headern ist damit zu rechnen, dass sich das in naher Zukunft ändert. Mehr Informationen finden Sie in aktuellen RFC-Draft [HTTPLink].

5.6 Zusammenfassung

In diesem Kapitel haben wir Verben (Methoden) betrachtet, die ein zentraler Bestandteil des REST-Konzepts sind und von allen Ressourcen, die in einer spezifischen Ausprägung des Architekturstils enthalten sind, gleichermaßen unterstützt werden sollten. Im HTTP-Umfeld sind dies die Standardmethoden aus der HTTP-Spezifikation, allen voran GET, HEAD, PUT, POST und DELETE.

Für den Entwurf Ihrer REST-Anwendungen gilt als Faustregel, dass Sie mit diesen Verben auskommen sollten. Die Verwendung von POST mit einer anderen Semantik als dem Anlegen neuer Ressourcen sollte dabei die Ausnahme sein. Auch wenn die Definition neuer Verben kein Regelfall ist, sollten Sie auch diese Möglichkeit des HTTP-Protokolls kennen, entsprechende Standardisierungen beobachten und sich im Zweifelsfall dieser Option bedienen.

Wir haben damit drei der wesentlichen Elemente von REST betrachtet: Ressourcen, Identifikation und Standardmethoden. Im nächsten Kapitel widmen wir uns dem wichtigsten Aspekt, der das Web erst zum Web macht: Hypermedia.