2 Einführung in REST

REST ist mittlerweile im »Mainstream« angekommen, und tatsächlich haben die unter diesem Namen zusammengefassten Konzepte das WWW, das wir heute kennen, schon vor der aktuellen Popularität bereits über lange Jahre geprägt. In diesem Kapitel werden wir uns daher zunächst mit der Geschichte von REST beschäftigen und danach die wichtigsten Grundprinzipien erläutern.

2.1 Eine kurze Geschichte von REST

Die Anfänge des Web in den frühen 90er-Jahren waren von einer sehr einfachen Metapher geprägt: Dokumente in einem Standardformat, die über eine eindeutige Identifikation verfügten und sich darüber gegenseitig referenzieren konnten, sowie ein einfaches Protokoll, um diese Dokumente zu übertragen. Schon bald jedoch wurde das Web mit dynamischen Informationen angereichert, die auf Datenbankinhalten oder mehr oder weniger komplexen Berechnungen beruhten.

Daraus resultierte eine Unklarheit in der Bedeutung der einzelnen Konzepte des WWW. Identifiziert eine URI ein bestimmtes Dokument oder einen Dienst, der viele Dokumente (oder allgemeiner: Informationen) liefern kann? Wie passt die Metapher eines Dokumentes, das transferiert wird, zu einem Dienst, der sein Ergebnis auf Basis einer komplexen Implementierung ermittelt? Dem Web in der Praxis fehlte eine zugrunde liegende Theorie, ein konzeptionelles Modell mit klaren Begriffen, mit dessen Hilfe man über das Pro und Contra von möglichen Weiterentwicklungen diskutieren konnte. Mit anderen Worten: Dem Web fehlte ein theoretisches Fundament – eine formale Architektur.

Fielding, der seit Beginn (ca. 1994) in die Standardisierung der Webprotokolle involviert war, konstruierte deshalb ein Konzept, das initial den Namen »HTTP Object Model« trug. Während der Standardisierung von HTTP 1.0, vor allem aber während der Weiterentwicklung zu HTTP 1.1 nutzte er dieses Modell zum einen als Rahmen für das Design von HTTP und passte es zum anderen an, wenn es einer sinnvollen Veränderung im Weg stand.

Im Kern dieses Modells steht die Idee, für statische Inhalte und dynamisch berechnete Informationen, die ein globales, gigantisches Informationssystem bil-

den, ein einheitliches Konzept zu definieren – eine Idee, mit dem wir uns in den weiteren Kapiteln detailliert beschäftigen werden.

In seiner Dissertation, die er im Jahr 2000 beendete, abstrahierte Fielding von der konkreten Architektur, die HTTP zugrunde liegt, und legte den Schwerpunkt auf die Konzepte anstatt auf die konkrete Syntax, die Details des Protokolldesigns und die vielen Kompromisse, die aus Kompatibilitätsgründen eingegangen werden mussten. Als Ergebnis entstand der Architekturstil[1] REST. Dieser ist somit eine Stufe abstrakter als die HTTP-Architektur: Theoretisch könnte man die Prinzipien von REST auch mit einem neu erfundenen Satz von Protokollen umsetzen. In der Praxis ist jedoch tatsächlich nur das Web als konkrete Ausprägung des Architekturstils relevant.

HTTP, URIs und die anderen Standards des Web lassen sich auf unterschiedliche Arten verwenden. Idealerweise setzt man eine Technologie so ein, wie es sich deren Architekt vorgestellt hat, denn dadurch ist die Wahrscheinlichkeit am höchsten, dass man auch die Vorteile optimal nutzt. Aber natürlich kann man auch gegen die Regeln verstoßen, und in der Praxis kommt das sehr häufig vor: Viele der Frameworks, Bibliotheken und öffentlichen Web-APIs sind alles andere als »RESTful«, auch wenn dies häufig behauptet wird.

Das Paradebeispiel für eine Verwendung der Protokolle des WWW ohne den Einsatz der entsprechenden Architektur sind Webservices auf Basis von SOAP und WSDL. Diese verwenden zwar HTTP, aber eines der Kernkonzepte besteht in der sogenannten »Transportunabhängigkeit«: HTTP ist nur einer von vielen Mechanismen, über den Daten von A nach B übertragen werden können.[2] Aus diesem Grund gab es in den vergangenen Jahren immer wieder Debatten darüber, ob nun SOAP oder REST die bessere Lösung sei[3]. Lange Zeit war REST dabei nur der Favorit einer kleinen Minderheit; eine zunehmende Frustration mit dem Webservices-Technologiestack und eine immer stärkere Verbreitung von Web-APIs ohne SOAP[4] haben jedoch mittlerweile zu einem großen Interesse an REST im Service-Umfeld geführt. In vielen Fällen wird REST dabei ohne größere Debatte als die »richtige« Lösung vorausgesetzt.[5]

Aber auch ganz ohne Webservices kann man mehr oder weniger katastrophal gegen die Grundregeln von REST verstoßen, sehr gut zu sehen an Webframe-

1. Wenn Ihnen der Begriff »Architekturstil« nichts sagt, Sie aber wissen, was »Design Patterns« (Entwurfsmuster) sind, können Sie sich einen Architekturstil sehr ähnlich vorstellen, nur auf der nächsthöheren Ebene. So wie ein Entwurfsmuster eine Stufe abstrakter ist als eine konkrete Implementierung, ist ein Architekturstil eine Stufe abstrakter als eine spezifische Architektur.
2. Diese Reduktion des Applikationsprokolls HTTP auf ein reines Transportprotokoll ist einer der Hauptkritikpunkte der REST-Verfechter am SOAP/WSDL/WS-*-Universum.
3. Genau genommen ein klassischer Äpfel- und Birnenvergleich: ein konkretes XML-Format gegen einen abstrakten Architekturstil, genau so klug wie der Vergleich von Spring mit dem Observer-Pattern.
4. Z. B. bei Google, Yahoo!, Facebook, Twitter und sogar Microsoft.
5. Diesen Absatz haben wir nicht ganz ohne Freude für jede Auflage des Buches überarbeiten und der jeweils deutlich gestiegenen Akzeptanz von REST anpassen müssen.

works bzw. damit erstellten Anwendungen, die sich nicht wie Webanwendungen anfühlen, nicht so entwickelt werden und auch keinen der Vorteile des Web ausnutzen. Während sich Benutzer mit defekten Back- und Forward-Schaltflächen, ablaufenden Sitzungen, schlechten Antwortzeiten und nicht verlinkbaren Informationen herumplagen, ist auch aus Architektursicht hier weder von REST noch von RESTful HTTP etwas zu sehen.

Eine kurze Anmerkung zu Unterscheidung »REST« und »RESTful HTTP«: Im Kontext dieses Buches verwenden wir den Begriff REST formal gesehen nicht immer korrekt. Genau genommen müssten wir konsequent unterscheiden zwischen dem abstrakten Architekturstil REST, der konkreten, weitgehend REST-konformen Implementierung HTTP und einzelnen Webanwendungen und Diensten, die mehr oder weniger konform zu den REST-Prinzipien umgesetzt sein können. Wir folgen jedoch stattdessen dem allgemeinen Sprachgebrauch und benutzen »REST« und »REST-konforme HTTP-Nutzung« (englisch »RESTful HTTP«) in den meisten Fällen synonym[6].

2.2 Grundprinzipien

Wenn wir REST auf ein Minimum reduzieren, ergeben sich fünf Kernprinzipien, die wir wie folgt zusammenfassen können:

- Ressourcen mit eindeutiger Identifikation
- Verknüpfungen/Hypermedia
- Standardmethoden
- Unterschiedliche Repräsentationen
- Statuslose Kommunikation

Schauen wir uns diese Prinzipien näher an.

Eindeutige Identifikation

Wenn Sie sich an das letzte System erinnern, das Sie entwickelt haben, so gab es sicherlich eine Menge von Kernabstraktionen, deren Instanzen es verdient hätten, eindeutig identifiziert zu werden. Im Web gibt es ein einheitliches Konzept für die Vergabe von IDs – nämlich die URI. URIs bilden einen globalen Namensraum, und die Verwendung einer URI als ID stellt sicher, dass Sie die wesentlichen Elemente weltweit eindeutig identifizieren können.

Der wesentliche Vorteil eines konsistenten Schemas für Namen ist, dass Sie kein eigenes mehr erfinden müssen – Sie können sich auf das bereits bestehende verlassen, das, wie man kaum bestreiten kann, sehr gut funktioniert, hervorragend skaliert und von wirklich jedem verstanden wird. Wenn Sie an ein zufällig

6. Hetzen Sie uns ruhig die @RESTPolice auf den Hals (ja, das ist ein real existierender Twitter-Account).

ausgewähltes Geschäftsobjekt aus Ihrer letzten Anwendung denken, können Sie wahrscheinlich leicht einen Fall konstruieren, in dem die Identifikation über URIs nützlich gewesen wäre. Hat Ihre Anwendung zum Beispiel eine Abstraktion »Kunde« enthalten, wären Ihre Anwender wahrscheinlich gern in der Lage gewesen, einem Kollegen einen Link auf einen bestimmten Kunden zu schicken, sich ein Lesezeichen zu setzen oder die ID (die URI) des Kunden einfach nur auf einem Stück Papier zu notieren. Und stellen Sie sich vor, wie viel Geschäft ein Onlineunternehmen wie Amazon.com nicht machen würde, wenn man keinen Link auf jedes einzelne Produkt verschicken könnte!

Wird man zum ersten Mal mit dieser Idee konfrontiert, fragt man sich, ob man nun jeden einzelnen Datenbankeintrag (und dessen ID) exponieren sollte – und schaudert beim bloßen Gedanken. Schließlich haben uns Jahre der Objektorientierung gelehrt, dass wir Implementationsdetails verbergen sollten. Aber dies ist nur ein scheinbarer Konflikt: Die Dinge (Ressourcen), die Sie nach außen bekannt geben, befinden sich auf einem anderen Abstraktionsniveau als die einzelnen Datenelemente. Ein Beispiel: Man würde in einem Bestellsystem vielleicht eine Ressource »Order« exponieren und ihr eine URI spendieren, nicht jedoch jeder einzelnen Bestellposition oder der Liefer- und Rechnungsadresse[7]. Treibt man das Prinzip, alle sinnvollen Dinge zu identifizieren, konsequent weiter, so ergeben sich auf einmal neue Ressourcen, an die man normalerweise nicht gedacht hätte – ein Prozess oder Prozessschritt, ein Verkauf, eine Verhandlung, eine Angebotsanfrage – alles Beispiele für Ressourcen, die es zu identifizieren lohnt. Dies wiederum kann dann dazu führen, dass es mehr persistente Entitäten gibt als in einem Nicht-REST-Design.

Im Folgenden einige Beispiel-URIs:

- `http://example.com/customers/1234`
- `http://example.com/orders/2007/10/776654`
- `http://example.com/products/`
- `http://example.com/processes/salary-increase-234`

Da wir menschenlesbare URIs verwendet haben – eine sinnvolle, wenn auch aus REST-Sicht völlig irrelevante Entscheidung –, sollten sie relativ leicht zu interpretieren sein. Die Vorteile des einheitlichen, global gültigen Namensschemas gelten sowohl für die browserbasierte als auch für die Anwendung-zu-Anwendung-Kommunikation.

Fassen wir das erste Prinzip noch einmal zusammen: Verwenden Sie URIs, um alle die Instanzen aller wesentlichen Abstraktionen Ihrer Anwendung zu identifizieren, die es Wert sind – unabhängig davon, ob es sich um individuelle Einträge oder Mengen handelt.

---

7. Welche Ressourcen Sie tatsächlich definieren, hängt natürlich stark vom Einzelfall ab.

Hypermedia

Das nächste Prinzip hat die formale Beschreibung »Hypermedia as the engine of application state«. Im Kern verbirgt sich dahinter das Konzept von Verknüpfungen (Links). Diese sind uns allen aus HTML bekannt, aber in keiner Weise auf menschliche Nutzer beschränkt. Am besten verdeutlicht dies ein Beispiel:

```
{
    "amount": 23,
    "customer": {
        "href": "http://example.com/customers/1234"
    },
    "product": {
        "href": "http://example.com/products/4554"
    },
    "cancel": {
        "href": "./cancellations"
    }
}
```

Wenn Sie sich die Produkt- und Kundenverknüpfungen in diesem Beispiel ansehen, können Sie sich leicht vorstellen, dass eine Applikation diesen Links »folgt«, um an weitere Informationen zu gelangen. Das wäre natürlich genauso gut denkbar, wenn nur ein einfaches ID-Attribut z. B. mit einem Integerschlüssel belegt wäre – allerdings nur im Kontext dieser einen Anwendung. Das Wunderbare am Hypermedia-Ansatz des Web ist, dass Verknüpfungen anwendungsübergreifend funktionieren – es kann sich eine andere Anwendung, ein anderer Server oder ein anderes Unternehmen auf einem anderen Kontinent dahinter verbergen, ohne dass sich der Konsument dafür interessieren muss. Weil das Namensschema global ist, können alle Ressourcen, aus denen das Web besteht, miteinander verknüpft werden.[8]

Wichtiger noch als die Verknüpfung zwischen Daten ist die Steuerung des Applikationszustands über Hypermedia: Ein Server kann dem Client über Hypermedia-Elemente wie z. B. Links mitteilen, welche Aktionen er als Nächstes ausführen kann. Abstrakt formuliert: Die Applikation wird damit von einem Zustand in den nächsten überführt, indem der Client einem Link folgt.

Ein Beispiel: Unsere Bestellung enthält einen Link, der eine Verknüpfung kapselt, die der Client zum Stornieren der Bestellung verwenden kann. Ist dies im aktuellen Status der Bestellung nicht mehr möglich, enthält die Bestellung diesen Link nicht mehr: Welche Statusübergänge zu einem beliebigen Zeitpunkt möglich sind, wird also durch Hypermedia gesteuert.

Aber nicht nur Links sind Hypermedia-Elemente. Diese sind nur ein Weg für einen Client, auf Basis von Informationen, die vom Server bereitgestellt werden,

8. Das Beispiel ist übrigens sehr ineffizient: In einem realistischen Szenario würde man auch einige wesentliche Attribute des Kunden und des Produktes mit in die Bestellung aufnehmen, um die Anzahl der Zugriffe zu minimieren.

herauszufinden, wie ein nächster Request konstruiert werden kann. Aus dem HTML-Umfeld bestens bekannt sind Formulare, HTML-Forms. Im Gegensatz zu Links, die ein explizites Ziel haben, also auf eine bestimmte Ressource verweisen, enthalten Formulare die Regeln, nach denen einer von potenziell beliebig vielen neuen Requests erzeugt werden kann. Das folgende Beispiel illustriert das an einer Kombination verschiedener Formularelemente:

```
<form action="/reports" method="GET">
  <select name="month">
    <option value="01">01</option>
    <option value="02">02</option>
    ...
    <option value="12">12</option>
  </select>
  <input type="number" name="year">
  <select name="state">
    <option></option>
    <option value="cancelled">cancelled</option>
    <option value="commited">commited</option>
    <option value="created">created</option>
    <option value="failed">failed</option>
    <option value="processing">processing</option>
    <option value="shipped">shipped</option>
  </select>
  <input type="submit" value="get report"/>
</form>
```

Die Menge an Kombinationsmöglichkeiten aus den beiden Select-Auswahlfeldern und dem Eingabefeld ist unendlich. Für den Client ist damit nur vorgegeben, wie er einen POST-Request konstruieren kann. Der Server ist dafür zuständig, diesen dann zu interpretieren und entweder mit einer direkten Antwort oder mit einer Weiterleitung auf eine andere, dem Client vorher unbekannte Ressource zu beantworten. Der Server steuert den Client höchst dynamisch über die Hypermedia-Elemente, die er in seine Antworten einbettet.

Zusammenfassung: Verwenden Sie Hypermedia, um Beziehungen zwischen Ressourcen herzustellen, wo immer es möglich ist, und steuern Sie den Applikationsfluss darüber. Hypermedia-Controls sind es, die das Web erst zum Web machen.

Standardmethoden

Bei der Diskussion der ersten beiden Prinzipien haben wir eine implizite Annahme unerwähnt gelassen, nämlich die Tatsache, dass eine nutzende Anwendung mit den URIs bzw. den Ressourcen, die durch sie identifiziert werden, tatsächlich etwas anfangen kann. Wenn Sie eine URI auf der Werbefläche eines Busses sehen und diese in die Adresszeile Ihres Browsers eintippen – woher weiß dieser, was er mit der URI tun kann?

Er weiß, was er damit tun kann, weil jede Ressource die gleiche Schnittstelle unterstützt, den gleichen Satz von Methoden[9]. Bei HTTP zählen zu diesen Methoden – zusätzlich zu den beiden, die jeder kennt (GET und POST) – auch PUT, DELETE, HEAD und OPTIONS. Die Bedeutung dieser Methoden oder »Verben« ist in der HTTP-Spezifikation beschrieben, inklusive einiger Garantien über ihr Verhalten. Als Analogie könnte man die HTTP-Schnittstelle mit einem Java-Interface vergleichen, das von jeder Ressource implementiert wird (in Pseudosyntax):

```
interface Resource {
  Resource(URI u);
  Response options();
  Response get();
  Response post(Request r);
  Response put(Request r);
  Response delete();
}
```

Listing 2–1 *Pseudocode für das REST-Interface*

Da für jede Ressource das gleiche Interface verwendet wird, können Sie mit Recht erwarten, dass Sie eine Repräsentation, eine Darstellung der Ressource, mit der GET-Methode abholen können. Da die Semantik von GET in der HTTP-Spezifikation beschrieben ist, können Sie sich darauf verlassen, dass Sie damit keine Verpflichtungen eingehen: Die Methode ist »safe« (sicher). Da GET ein sehr effizientes und raffiniertes Caching unterstützt, muss der Request in vielen Fällen gar nicht zum Server geschickt werden. Sie können sich ebenfalls darauf verlassen, dass die Methode GET idempotent ist. Das bedeutet, dass Sie im Fall einer ausbleibenden Antwort die Anfrage einfach noch einmal schicken können (ob die erste Anfrage nun angekommen ist oder nicht). Idempotenz ist ebenfalls garantiert für die nicht sicheren Methoden PUT (ein Aktualisieren oder Erzeugen, je nachdem, ob die Ressource vorher schon existierte oder nicht) und DELETE (ein Löschen, das ein zweites Mal ohne Effekt bleibt). Nur für POST, das entweder eine neue Ressource erzeugt oder eine beliebige Verarbeitung anstößt, gibt es keine Garantien. Natürlich sind all diese Garantien aus der Spezifikation nichts wert, wenn die an der Kommunikation beteiligten Partner sie nicht erfüllen. Browser tun das sehr konsequent, für die Serverseite sind – bei einer Webanwendung – Sie verantwortlich.

Aber auch wenn Sie die Funktionalität Ihrer Anwendung für andere Anwendungen in einer REST-konformen Art und Weise nutzbar machen wollen oder eine solche Funktionalität nutzen wollen, gelten dieses Prinzip und seine Restriktionen ebenfalls. Für jemanden, der einen Entwurfsansatz gewohnt ist, bei dem

9. Wobei mit »unterstützen« gemeint ist, dass Sie die Methode auf jeden Fall aufrufen können – möglicherweise wird sie von einer spezifischen Ressource nicht unterstützt und Sie erhalten eine entsprechende Fehlermeldung.

man beliebig viele anwendungsspezifische Operationen erfinden kann, ist das schwer zu akzeptieren. Wie soll man allein mit GET, PUT, POST und DELETE auskommen? Es handelt sich dabei allerdings nur scheinbar um ein Problem: Vermeintlich anwendungsspezifische Methoden werden auf die Standard-HTTP-Methoden abgebildet, und um die Mehrdeutigkeit aufzulösen, wird ein ganzes Universum neuer Ressourcen angelegt.

Ein Trick? Nein – ein »GET« auf eine URI, die einen Kunden eindeutig identifiziert, ist genauso bedeutungsvoll wie eine Operation mit dem Namen »getCustomerDetails«. Gelegentlich benutzt man ein Dreieck, um diesen Effekt darzustellen[10]:

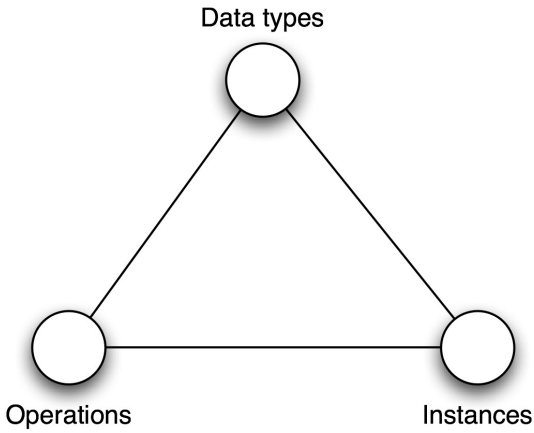

Abb. 2–1 *Das »Options-Dreieck«*

Stellen Sie sich die Eckpunkte als Drehregler vor, mit denen Sie beim Modellieren Ihres Systems die Anzahl seiner konzeptuellen Bestandteile anpassen können, bis Ihre Anforderungen vom Systemkonzept abgedeckt werden: Die Anzahl der Operationen, die Ihre Schnittstellen unterstützen, die unterschiedlichen Datentypen, die verwendet werden können, und die Anzahl der Instanzen (»Services« oder »Fassaden« bzw. Ressourcen). Bei dem Ansatz, den Sie gewohnt sind, drehen Sie an der »Operations«- und der »Data types«-Schraube, die Anzahl der Instanzen halten Sie typischerweise konstant (i.d.R. haben Sie so viele Instanzen, wie Sie unterschiedliche Services haben). Beim REST-Ansatz halten Sie die Anzahl der Operationen konstant und benutzen die anderen beiden Regler. Was wir damit sagen möchten: Sie können alles, was Sie mit dem einen Ansatz abbilden, auf den anderen übertragen – nur die Mittel ändern sich.

Dieser Aspekt, entscheidet, ob Ihre Anwendung Teil des Web ist oder nicht: Der Beitrag, den sie zu dessen Wert leistet, ist direkt proportional zur Anzahl der

10. Der Ursprung dieser Darstellung ist nicht ganz klar, aber vermutlich wurde sie zunächst von Benjamin Carlyle verwendet [Carlyle2008].

Ressourcen, die sie ihm hinzufügt. In einem REST-konformen Ansatz fügen Sie über Ihre Anwendung dem Web vielleicht eine Million Kunden-URIs hinzu. Wählen Sie stattdessen den althergebrachten CORBA-Stil, wie er bei SOAP-Services auch zum Einsatz kommt, liegt der Wertbeitrag bei einer URI, einem »Endpunkt« – vergleichbar einer Tür, die den Eintritt in ein Universum von Ressourcen nur denjenigen gestattet, die den Schlüssel dazu haben. Analog verhält es sich bei einer Anwendung, die Sie vom Browser aus nutzen und in der sich – obwohl Sie fleißig mit ihr interagieren – die in der Adresszeile angezeigte URI nie ändert: Auch hier haben Sie nur die Anwendung als Ganzes mit einer URI ausgestattet, nicht die vielen sinnvollen, einzelnen Abstraktionen, die sich in ihr verbergen.

Das Standardinterface ermöglicht es allen Komponenten, die das HTTP-Anwendungsprotokoll verstehen, mit Ihrer Applikation zu kommunizieren. Beispiele solcher Komponenten sind außer dem Browser auch andere generische Clients wie curl oder Wget, Proxy-Server, Caches, HTTP-Server, Gateways oder auch Dienste wie Google, Yahoo!, Bing usw.

Zusammenfassung: Damit die Clients, von denen Sie wissen, ebenso wie diejenigen, von denen Sie nichts ahnen, mit Ihren Ressourcen kommunizieren können, sollten alle Ressourcen das Standardanwendungsprotokoll (HTTP) korrekt implementieren.

Ressourcen und Repräsentationen

Eine kleinere Komplikation haben wir bislang ignoriert: Wie weiß ein Client, was er mit den Daten machen soll, die er z. B. als Ergebnis eines GET- oder POST-Requests erhält? Der Ansatz von HTTP ist, eine Trennung der Verantwortlichkeiten für Daten und Operationen einzuführen. Mit anderen Worten: Ein Client, der ein bestimmtes Datenformat verarbeiten – d. h. lesen oder erzeugen – kann, ist in der Lage, mit jeder Ressource zu interagieren, die eine Repräsentation in diesem Format zur Verfügung stellen kann. Die Operationen sind schließlich überall die gleichen. Sehen wir uns dazu wiederum ein Beispiel an. Auf Basis von HTTP Content Negotiation kann ein Client eine Repräsentation in einem bestimmten Format anfordern:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml
```

Ergebnis könnte hier ein unternehmensspezifisches XML-Format sein, das für Kundeninformationen verwendet wird. Alternativ könnte der Client folgende Anfrage senden:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

In diesem Fall könnte eine Kundenadresse im vCard-Format zurückgeliefert werden. (Die Antworten sind nicht dargestellt, sie würden Metadaten über das Datenformat im HTTP Content-Type-Header enthalten.) Dies zeigt auch, warum für Repräsentationen idealerweise Standardformate zum Einsatz kommen sollten: Kennt ein Client sowohl das Standardformat als auch das HTTP-Protokoll, kann er mit jeder beliebigen Ressource auf der Welt interagieren, für die eine Repräsentation in diesem Format abrufbar ist. Unglücklicherweise gibt es nicht für alle Anwendungsfälle ein Standardformat, aber Sie können sich sicher vorstellen, wie ein kleineres Ökosystem entstehen könnte, z. B. innerhalb eines Unternehmens oder zwischen Partnern. Natürlich beziehen sich all diese Aussagen nicht nur auf die Daten, die vom Server zum Client gesendet werden, sondern auch auf die Gegenrichtung – jeder Server, der Daten in einem bestimmten Format akzeptieren kann, interessiert sich nicht für die Details des Clients, solange dieser dem Applikationsprotokoll folgt.

Es gibt einen weiteren wichtigen Vorteil durch mehr als eine Repräsentation pro Ressource: Stellen Sie sowohl eine HTML- als auch eine XML-Repräsentation Ihrer Ressourcen zur Verfügung, sind diese nicht nur von Ihrer Clientapplikation, sondern durch jeden Standardwebbrowser darstellbar. Mit anderen Worten: Die Informationen in Ihrer Applikation sind nun für jeden zugänglich, der weiß, wie man mit dem Web umgeht.

Es gibt noch einen anderen Weg, diese Tatsache auszunutzen: Ihre Webanwendung wird zu Ihrem Web-API – schließlich ist das API-Design häufig von der Idee getrieben, dass alles, was man über das User Interface (UI) tun kann, auch über das API möglich sein sollte. Beide Aufgaben zu einer einzigen zu verschmelzen ist ein erstaunlich eleganter Weg, um eine bessere Webschnittstelle sowohl für menschliche als auch maschinelle Nutzer zu erstellen.

Zusammenfassung: Stellen Sie unterschiedliche Repräsentationen Ihrer Ressourcen für unterschiedliche Anforderungen zur Verfügung.

Statuslose Kommunikation

Das fünfte und letzte Prinzip ist die statuslose Kommunikation. »Statuslos« bezieht sich hier nicht etwa auf den serverseitigen Zustand – natürlich kann (und soll) selbiger Zustand halten. Aber REST schreibt vor, dass dieser Zustand entweder vom Client gehalten oder vom Server in einen Ressourcenstatus umgewandelt wird. Nicht gewollt ist ein serverseitig abgelegter, transienter, clientspezifischer Status über die Dauer eines Requests hinweg. Anders formuliert: Der Server interessiert sich für den Client nur, wenn er gerade einen seiner Requests verarbeitet, und kann seine Existenz direkt danach wieder vergessen.

Sie müssen den Zustand also entweder vollständig auf dem Client halten, indem Sie ihn als Teil der Repräsentation vom Server zum Client übermitteln, oder aber in einen Ressourcenstatus umwandeln. Ein Beispiel: Ihr Warenkorb – das klassische Beispiel für einen Sitzungsstatus – wird vielleicht zu einer eigenen

Ressource, mit dem Vorteil, dass Sie ein Lesezeichen darauf setzen und einen Link per E-Mail versenden können.

Durch den Verzicht auf einen Sitzungsstatus wird die Kopplung des Clients an den Server verringert, da zwei aufeinanderfolgende Anfragen nicht von der gleichen Serverinstanz bearbeitet werden müssen, wodurch die Skalierbarkeit vereinfacht wird. Die Kopplung wird auch aus anderer Sicht verringert: Ein Client könnte ein Dokument mit Verknüpfungen vom Server beziehen und verarbeiten, während der Server heruntergefahren, Hardware ausgetauscht, das Betriebssystem aktualisiert und das System wieder hochgefahren wird. Folgt der Client zu einem späteren Zeitpunkt einem Link, ist das, was in der Zwischenzeit passiert ist, irrelevant.

2.3 Zusammenfassung

In diesem Kapitel haben Sie den wesentlichen Unterschied zwischen REST als abstraktem Architekturstil und HTTP, URI und den anderen Webstandards als einer konkreten Ausprägung dieses Stils kennengelernt. Darüber hinaus kennen Sie nun die wesentlichen Grundkonzepte: anwendungsübergreifend standardisierte Identifikation, Hypermedia, eine Schnittstelle mit einer fest definierten Menge von Operationen, Ressourcenrepräsentationen und statuslose Kommunikation. Im nächsten Kapitel werden wir uns ansehen, wie man einen konkreten Anwendungsfall auf diese Prinzipien abbilden kann.