8 Grundlagen RESTful HTTP

Mit diesem Kapitel verlassen wir die Programmiersprachen-APIs und kommen zu den Remote-APIs, die eine explizite Grenze und häufig auch Interoperabilität zwischen API-Konsument und API-Anbieter gewährleisten, sodass Systeme unterschiedlichster Plattformen zusammen funktionieren können. Wer APIs bauen möchte, die tatsächlich die Bezeichnung »Webservice« verdienen, muss den Architekturstil des Web berücksichtigen. Deswegen bietet Ihnen dieses Kapitel wichtige Grundlagen über REST und HTTP.

8.1 REST versus HTTP

Das Akronym REST steht für REpresentational State Transfer und wird in der Dissertation von Roy Fielding [Fielding 2000] erstmalig beschrieben. REST ist weder eine konkrete Technologie noch ein offizieller Standard. Es handelt sich vielmehr um einen Softwarearchitekturstil, bestehend aus Leitsätzen und bewährten Praktiken für netzwerkbasierte Systeme.

REST und HTTP werden häufig in einem Atemzug genannt. Das liegt daran, dass REST typischerweise mit HTTP umgesetzt wird. In diesem Fall spricht man von RESTful HTTP [Tilkov et al. 2015]. Doch nicht jede API, die HTTP verwendet, ist automatisch REST-konform. Außerdem ist HTTP nicht das einzige Protokoll, mit dem REST-konforme Applikationen realisiert werden können.

HTTP überträgt die Daten auf die Anwendungsschicht und ist vermutlich der wichtigste Standard im Web. HTTP basiert auf einem einfachen Request-Response-Modell, bei dem Nachrichten zwischen einem Client und einem entfernten Server über ein Netzwerk ausgetauscht werden. Ein Server kann eine Response-Nachricht nur als Antwort auf eine Request-Nachricht versenden.

In dieser Einführung werden die formalen Bedingungen des Architekturstils REST etwas vereinfacht, weil für dieses Buch nur RESTful

HTTP interessant ist. Andere Umsetzungen der REST-Prinzipien werden nicht betrachtet. Daher soll nachfolgend REST immer im Sinne von RESTful HTTP verstanden werden.

8.2 REST-Grundprinzipien

Die Grundprinzipien von REST kann man laut Stefan Tilkov et al. [Tilkov et al. 2015] folgendermaßen zusammenfassen:

- Eindeutige Identifikation von Ressourcen
- Verwendung von Hypermedia
- Verwendung von HTTP-Standardmethoden
- Unterschiedliche Repräsentationen von Ressourcen
- Statuslose Kommunikation

Was genau hinter diesen Grundprinzpien steht, wird im Einzelnen in den folgenden Abschnitten beschrieben.

Eindeutige Identifikation von Ressourcen

Jede Ressource braucht zur Identifikation einen eindeutigen Schlüssel. Stellen Sie sich vor, Sie könnten Videos auf YouTube, Tickets in Jira oder Produkte bei Amazon nicht eindeutig über Links identifizieren. Das wäre ziemlich unpraktisch, oder? Für das Web eignen sich für diese Aufgabe Uniform Resource Identifiers (URIs). Diese einheitlichen Bezeichner für Ressourcen bilden einen globalen Namensraum. Hier sind einige Beispiele:

```
http://example.com/answers/42
http://example.com/agents/agent-007/cars
http://example.com/users?locked=true
```

Man kann annehmen, dass die erste URI im obigen Beispiel genau eine Entität identifiziert. Es könnte die Antwort mit der ID 42 gemeint sein. Derartige menschenlesbare URIs werden nicht per se von REST verlangt, vereinfachen aber die Arbeit mit APIs. Auch die beiden anderen Beispiele sind intuitiv verständlich. Das zweite Beispiel identifiziert kein einzelnes Fahrzeug, sondern alle von Agent 007. Im dritten Beispiel werden alle gesperrten Benutzer selektiert.

In den letzten beiden Beispielen werden Collections identifiziert. Das ist jedoch kein Widerspruch zur Forderung, dass jede Ressource eine eindeutige Identifikation haben soll, denn Collections sind ebenfalls Ressourcen.

URI versus URL

URIs und URLs sind zusammen in RFC 3986 [Fielding & Masinter 2005] definiert. In den meisten Fällen (wie auch in diesem Buch) können URIs und URLs synonym verwendet werden, denn jede URL ist per Definition auch eine URI. Das gilt aber nicht umgekehrt: Nicht jede URI ist auch eine URL. Deswegen soll an dieser Stelle einmal deren Unterschied erklärt werden.

Eine URL ist eine kurze Zeichenkette, die eine Ressource identifiziert. Eine URI ist ebenfalls eine kurze Zeichenkette, die eine Ressource identifiziert. Was ist der Unterschied?

Eine URI ist in erster Linie ein eindeutiger Identifikator und es gibt keine Garantie, dass für diesen eine Repräsentation existiert. Ein XML-Namensraum heißt typischerweise nicht »User«, weil diese Bezeichnung höchstwahrscheinlich nicht eindeutig wäre. Auch andere Entwickler könnten diesen Namen verwenden. Folglich wählt man einen Namen wie »http://mycompany.com/person«. Diese Bezeichnung ist eindeutig und ein Konflikt mit anderen Namensräumen sehr unwahrseinlich. Der Name des XML-Namensraums ist somit eine URI, aber ist der Name auch eine URL? Nein. Denn es gibt nicht notwendigerweise eine Repräsentation, die man unter dieser URI aufrufen könnte.

Übrigens ist auch eine URN (Uniform Resource Name) eine URI [Berners-Lee et al. 1998], die dauerhaft und ortsunabhängig nach dem urn-Schema eine Ressource identifiziert. Mit der ISBN eines Buches kann beispielsweise eine URN für das Buch gebildet werden. Sie enthält auch keine Protokollangaben (http, ftp, amqp etc.), mit der ein Client versuchen könnte, sie aufzurufen.

Verwendung von Hypermedia

Der Begriff Hypermedia ist eine Mischung aus »Hypertext« und »Multimedia«. Das griechische Präfix »hyper« (ὑπέρ) kann mit »über« und »hinaus« übersetzt werden. Daher ist mit Hypertext ein Text gemeint, der über sich selbst hinaus weist. Hypermedia ist ein Oberbegriff von Hypertext. Während also Hypertext ausschließlich Texte verknüpft, schließt Hypermedia auch Dokumente, Bilder, Töne, Videos und andere multimedialen Inhalte mit ein.

Ein wesentliches Merkmal von Hypermedia stellen Links zur Verknüpfung verschiedener Medien dar. Links sind vor allem durch HTML sehr gut bekannt. Mit einem Browser kann man den Links leicht folgen. Sie dienen zur Ausführung von Aktionen und zur Navigation im Web. HTML ist daher ein klassisches Hypermediaformat. Die für den Client möglichen Aktionen und Navigationspfade werden vom Server über Hypermedia angeboten.

Aber selbstverständlich können auch andere Formate mit Links verbunden werden. Das folgende XML-Beispiel zeigt eine Nachricht,

deren Anhang nicht in die Nachricht eingebettet ist, sondern mittels einer URI referenziert wird.

```
<message self="http://example.com/messages/17">
  <body>...</body>
  <attachment ref="http://example.com/attachments/1701" />
</message>
```

Der Empfänger dieser Nachricht kann leicht dem Link folgen und weitere Informationen erhalten. Der entscheidende Vorteil der URIs ist, dass man auch Ressourcen in anderen Systemen referenzieren kann.

Verwendung von HTTP-Standardmethoden

Die zuvor genannten Vorteile setzen voraus, dass die Clients wissen, wie sie die URIs korrekt aufrufen können. Deswegen brauchen Links nicht nur eine URI, sondern auch eine einheitliche Schnittstelle, deren Semantik und Verhalten allen Clients bekannt ist. Und genau das ist der Fall bei HTTP.

Die Schnittstelle der URIs besteht im Wesentlichen aus den HTTP-Methoden GET, HEAD, POST, PUT und DELETE. Ihre Bedeutung und Garantien sind in der HTTP-Spezifikation definiert. Weil diese allgemeine Schnittstelle für jede Ressource verwendet wird, kann man ohne spezielles Vorwissen mit einem einfachen GET eine Repräsentation abrufen. Diese Vorhersagbarkeit entspricht den beschriebenen Qualitätsmerkmalen aus Kapitel 2.

GET ist beispielsweise sicher, sodass Clients keine Seiteneffekte zu erwartet haben, wenn sie diese HTTP-Methode wählen. Denn ein rein lesender Service ändert nicht den Zustand der Daten.

Schauen wir uns nun an, wie man einen Service zur Benutzerverwaltung mit REST und HTTP entwerfen könnte. In der folgenden Tabelle wird die objektorientierte Schnittstelle auf die Ressource users abgebildet. Nicht alle HTTP-Methoden kommen hierfür zur Anwendung.

Tab. 8–1 *RESTful HTTP für objektorientierte Schnittstelle*

Objektorientierte Schnittstelle	RESTful HTTP
getUsers()	GET /users
updateUser()	PUT /users/{id}
addUser()	POST /users
deleteUser()	DELETE /users/{id}
getUserRoles()	GET /users/{id}/roles

Unterschiedliche Repräsentationen von Ressourcen

Mit den bisher genannten Eigenschaften von REST können Clients eine Ressource über deren URI identifizieren und mit den bekannten HTTP-Methoden aufrufen. Doch woher wissen die Clients, wie sie mit den zurückgegebenen Daten umgehen sollen. Die Lösung für dieses Problem ist recht einfach. Die Clients geben die Formate an, die sie bei der Kommunikation verwenden wollen.

Durch HTTP Content Negotiation können Clients Repräsentationen in bestimmten Formaten abfragen. Wenn ein Browser beispielsweise einen Request absendet, teilt er im Accept-Header dem Server mit, welche Medienformate (MIME-Types) er erwartet:

```
GET /soccerstats HTTP/1.1
Host: example.com
Accept: text/html, application/xhtml+xml,
        application/xml;q0.9, */*;q=0.8
```

In dieser Abfrage gibt der Browser gleich mehrere alternative Medienformate an. Standardmäßig hat jedes Format die Präferenz 1. Mit dem Parameter q kann die Präferenz auch explizit auf einen Wert zwischen 0 und 1 gesetzt werden. Konkret heißt das für unser Beispiel, dass der Browser HTML oder XHTML erwartet. Falls der Server diese Formate für die angegebene Ressource nicht anbieten kann, soll er XML verwenden. Falls auch das nicht möglich ist, soll er irgendein Format auswählen.

Browser können unzählige Ressourcen dank einheitlicher URIs und des HTTP-Standardanwendungsprotokolls abrufen und darstellen, sofern die Repräsentationen der Ressourcen in Standardformaten bereitgestellt werden können. Leider gibt es nicht für jede Anwendung und für jede Art von Client ein passendes Standardformat. Innerhalb eines Unternehmens oder zwischen Partnern können daher auch andere Formate vereinbart werden. Diese Formate können beispielsweise auf XML oder JSON basieren.

Medientypen (MIME-Types)

Die Abkürzung MIME steht für Multipurpose Internet Mail Extensions. Wie der Name verrät, war dieser Standard ursprünglich für E-Mails mit Anhängen gedacht. Denn Empfängern muss schließlich per Konvention mitgeteilt werden, welche Datentypen und Zeichencodierungen die einzelnen Teile einer E-Mail haben. MIME-Types erwiesen sich für E-Mails als sehr nützlich und fanden ebenfalls Anwendung im übrigen Web. So wird beispielsweise einem Browser bei einer HTTP-Übertragung mitgeteilt, in welchem Format der Webserver die Daten sendet. Diese Technik findet auch bei REST Anwendung: Ser-

→

ver verwenden die Medientypen, um das Format ihrer Antworten zu beschreiben. Clients nutzen die Medientypen, um dem Server mitzuteilen, welche Formate sie bevorzugen.

Die Namen der Medientypen sind nach einem einheitlichen Schema aufgebaut. Zu den Top-Level-Typen zählen zum Beispiel application, audio, image, text und video. Danach folgt ein Subtyp und weitere optionale Parameter. Zusammen ergibt das beispielsweise:

```
text/plain; charset=utf-8
```

Der Top-Level-Typ application wird für Daten verwendet, die nur von bestimmten Programmen verarbeitet werden können. Hierzu zählen beispielsweise PDF-Dokumente oder applikationsspezifische XML- und JSON-Formate.

Die Angabe der Medientypen (MIME-Types) ist in RFC 2046 [Borenstein 1996] spezifiziert und wird beispielsweise für den Content-Type-Header verwendet.

Statuslose Kommunikation

Das letzte Grundprinzip, das in dieser Einführung nicht fehlen darf, ist die statuslose Kommunikation. Konsequenterweise gibt es bei REST-konformen Anwendungen keinen Sitzungsstatus, der serverseitig über mehrere Clientanfragen hinweg vorgehalten wird. Stattdessen muss der Kommunikationszustand im Client oder in der Repräsentation der Ressource gespeichert werden. Hierdurch wird die Kopplung zwischen Client und Server verringert.

Diese Einschränkung bietet viele Vorteile: Zum Beispiel könnte ein Server zwischen zwei Requests mit aktualisierter Software neu gestartet werden. Der Client würde es nicht merken. Genauso gut könnte zur Lastverteilung ein Load Balancer die Requests zu unterschiedlichen Serverinstanzen routen. Auch aufeinanderfolgende Requests eines Clients könnten von unterschiedlichen Serverinstanzen bearbeitet werden. Auf Sticky-Sessions muss nicht geachtet werden. Statuslose Kommunikation ist ebenfalls eine Voraussetzung dafür, dass jede URI als Einsprungspunkt dienen kann.

8.3 Ressourcen – die zentralen Bausteine

Nach der Einführung in die Grundprinzipien von REST konzentriert sich dieser Abschnitt auf Ressourcen, die als Bausteine für eine RESTful API dienen.

Beim Entwurf einer API findet man die Ressourcen in der Regel schnell, wenn man die fachliche Domäne betrachtet. Gute Kandidaten sind die fachlichen Kernkonzepte der Applikation, für die eine API entworfen werden soll. Zum Beispiel eignen sich für die API eines Onlineshops die Konzepte Order, Product und Customer. Diese identifizierbaren Konzepte bilden dann die Ressourcen der Schnittstelle. Ein Rückschluss auf die Implementierung der Schnittstelle ist damit jedoch nicht möglich. Die Unterscheidung zwischen dem Ressourcenmodell einer Web-API einerseits und andererseits dem Domänenmodell der internen Implementierung der Applikation ist sehr wichtig. Das Ressourcenmodell der Web-API stellt einen Vertrag mit der Außenwelt dar.

Ressourcen und ihre Repräsentationen

Um einen Blick auf den Zustand einer Ressource werfen zu können, bedarf es einer Repräsentation. Eine solche Repräsentation ist stets eine von vielen möglichen Sichten oder Darstellungen einer Ressource. Mit einer Repräsentation kann eine Ressource in der Außenwelt dargestellt und verarbeitet werden. Typischerweise werden im Web die Formate HTML und XHTML verwendet. Für APIs wird hingegen häufig JSON und XML eingesetzt. Alle Repräsentationen sind gleichermaßen gültig. Es gibt demzufolge nicht die eine »richtige« Repräsentation einer Ressource. Durch Content Negotiation können sich Client und Server auf ein Format einigen. Dazu teilt der Client mithilfe des Accept-Headers dem Server mit, welche Formate er bevorzugt. Der Server muss dies nicht beachten, sollte es aber, sodass jeder Client die gewünschte Ressource im passenden Format bekommt.

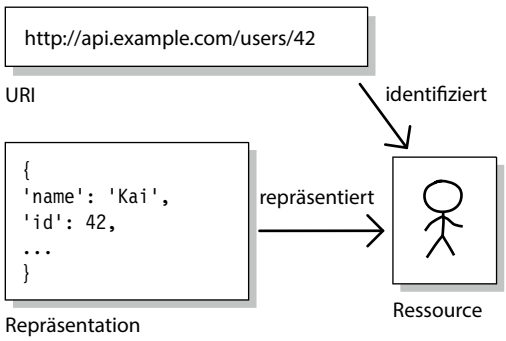

Abb. 8–1
Zusammenhang zwischen URI, Repräsentation und abstrakter Ressource

Die Unterscheidung von Ressource und Repräsentation ist manchmal nicht ganz offensichtlich. Denn nicht immer kann eindeutig entschieden werden, ob zwei unterschiedliche Repräsentationen zur gleichen Ressource gehören sollen oder zu verschiedenen. Denken Sie beispielsweise an ein Foto eines Hauses und dessen Grundriss. In beiden Fällen handelt es sich um Darstellungen desselben Hauses. In diesem speziellen Fäll würde man sich sicherlich für zwei unterschiedliche Ressourcen entscheiden, da der Grundriss andere Informationen enthält als das Foto. Umgekehrt könnte es aber für das Foto und den Grundriss jeweils unterschiedliche Formate geben.

Ressourcenkategorien

Fielding unterteilt Ressourcen nicht in Kategorien, denn aus REST-Sicht ist auch eine Liste von Entitäten eine Ressource. Dennoch ist es durchaus sinnvoll, die folgenden Kategorien einzuführen, weil ihre Unterscheidung für die Modellierung von REST-APIs hilfreich ist:

- Einzelressource
- Collection-Ressource
- Primärressource
- Subressource

Singular vs. Plural

Eine Ressource kann sich entweder auf eine einzelne Entität oder auf eine Collection von Entitäten beziehen. In beiden Fällen handelt es sich um Ressourcen im Sinne von REST. Als Name für eine Ressource sollte ein Substantiv verwendet werden, das die Bedeutung der Ressource treffend beschreibt. Es ist üblich, die Namen der Ressourcen einheitlich im Plural zu verwenden und nicht mit Singularvarianten zu mischen. Zur Identifikation eines einzelnen Objektes wird einfach deren ID an die URI angehängt.

http://api.example.com/products	Identifiziert eine Liste mit mehreren Produkten
http://api.example.com/products/42	Identifiziert das Produkt mit der ID 42

Dieses Schema ist einfach, aber ausreichend für viele unterschiedliche Anwendungsfälle. Es gibt jedoch auch Ausnahmen von dieser Regel: Ein gutes Beispiel wäre die Ressource »status«, mit der der Zustand eines Systems über die API exponiert werden könnte. In diesem besonderen Fall gibt es keine Pluralform, und gäbe es sie, würde sie fachlich keinen Sinn machen. Aber abgesehen von diesen Sonderfällen sollten die Namen von Ressourcen stets Substantive in Pluralform sein.

Schachtelung

Die Unterscheidung zwischen Primärressourcen und Subressourcen wird klar, wenn man das Thema Schachtelung betrachtet. Ressourcen auf oberer Ebene heißen Primärressourcen. Die Ressourcen auf den tieferen Ebenen sind die Subressourcen.

Ressourcen können untereinander in Beziehung stehen. Beziehungen mit den Kardinalitäten 1:1 und 1:n können durch Schachtelung abgebildet werden. Wenn zum Beispiel ein Kunde mehrere Adressen hat, könnten die Adressen als Subressource modelliert werden. Subressourcen können weitere Subressourcen enthalten, sodass die Schachtelung beliebig fortgesetzt werden kann.

```
/customers/customer-0815/addresses
```

Eine Ressource könnte prinzipiell durch mehrere URIs identifiziert werden. Bezogen auf das vorherige Beispiel könnte eine Adresse sowohl eine Subressource eines bestimmten Kunden als auch eine Subressource einer Addressliste sein.

```
/customers/customer-0815/addresses/address-01
/addresses/xtz381f81d
```

8.4 HTTP-Methoden

API-Designer und Softwarearchitekten sollten die Semantik der HTTP-Methoden genau kennen und beim API-Entwurf berücksichtigen. Das Web verlässt sich auf diese Regeln. Wer sie nicht einhält, muss mit unerwarteten Ergebnissen rechnen. Beispielsweise ist GET laut HTTP-Spezifikation eine sichere Methode. Wer diese Bedingung ignoriert, wird nach Murphys Gesetz[1] früher oder später ein Problem mit dieser Entscheidung haben. Laut der HTTP-Spezifikation ist allein der Server für eventuelle Seiteneffekte bei sicheren Methoden verantwortlich.

HTTP bietet die Methoden GET, HEAD, PUT, POST, DELETE, OPTIONS, TRACE und CONNECT [Fielding et al. 1999]. In RFC 5789 [Dusseault & Snell 2010] wird außerdem die Methode PATCH für HTTP definiert. Die für Web-APIs geläufigsten Methoden werden in den folgenden Abschnitten näher vorgestellt.

GET

Keine Zustandsänderungen

GET ist die wohl wichtigste und am häufigsten verwendete Methode im Web. Mit dieser Leseoperation kann die Repräsentation einer Ressource abgerufen werden. Laut HTTP-Spezifikation ist GET sicher und

1. Die bekannte Lebensweisheit des Ingenieurs Edward A. Murphy besagt, dass alles, was schiefgehen kann, auch schiefgehen wird.

idempotent. Deswegen kann GET aufgerufen werden, ohne dass Seiteneffekte zu befürchten sind. Wenn ein Client GET wählt, möchte er dem Server mitteilen, dass er keine Zustandsänderungen beabsichtigt. Sicherlich könnte der Server den Request loggen, doch der Zustand der Ressource sollte davon nicht betroffen sein.

Conditional GET

Mit einem bedingten GET kann der Client dem Server mitteilen, in welchen Fällen er die Daten der Ressource benötigt. Beispielsweise möchte der Client die Daten nur haben, falls sich diese seit einem bestimmten Zeitpunkt verändert haben. HTTP bietet für diesen Zweck den If-Modified-Since-Header. Falls sich die Daten seit dem angegebenen Zeitpunkt nicht geändert haben, antwortet der Server mit dem Statuscode 304 »Not Modified«. Alternativ kann ein bedingtes GET mit einem If-None-Match-Header ausgeführt werden. Der If-None-Match-Header ist ein genauerer Ersatz für den If-Modified-Since-Header und wird deswegen bevorzugt vom Server genutzt. Weitere Informationen zum Caching bietet Abschnitt 13.6.

»Not Modified«

Partial GET

Mithilfe des Range-Header-Feldes können partielle GET durchgeführt werden. Nur ein Teil der Entität wird hierbei übertragen.

HEAD

Die Methode HEAD ist genauso wie GET idempotent und sicher. Laut Spezifikation muss der Server für HEAD und GET die gleichen Metadaten liefern. Trotz dieser Gemeinsamkeiten gibt es einen wichtigen Unterschied: Eine Response auf HEAD darf keinen Nachrichtenrumpf haben.

Reduktion des Overheads

Durch die Reduktion des Kommunikationsaufwands kann die Performance einer Anwendung verbessert werden, denn nicht für jeden Request muss tatsächlich eine Repräsentation der Ressource übertragen werden: Ein Client kann mit HEAD beispielsweise prüfen, ob eine Ressource überhaupt existiert. Außerdem kann ein Client den Zeitpunkt der letzten Änderung herausfinden. Soll beispielsweise ein Video geladen werden, könnte der Client zunächst mit HEAD dessen Größe abfragen. Das Video wird dabei noch nicht übertragen.

```
HEAD /videos/1234.mp4 HTTP/1.1
Host: example.com
```

Die Antwort des Servers enthält nur Metadaten. Der Client weiß nun, wie groß das Video ist. Die Größe wird hier in Bytes angegeben:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 32768
Content-Type: video/mp4
```

Mit diesen Informationen kann der Client den Download des Videos starten. Der folgende partielle GET-Request lädt die ersten 16384 Bytes. Beachten Sie, dass die Bereichsangabe mit dem Index 0 beginnt:

```
GET /pictures/123 HTTP/1.1
Host: example.com
Range: bytes=0-16383
```

Die Antwort enthält schließlich die Binärdaten:

```
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Content-Length: 32768
Content-Range: bytes 0-16383/32768
Content-Type: image/jpeg
{Binärdaten}
```

PUT

PUT ist das Gegenstück zur GET-Methode. Mit PUT wird entweder eine existierende Ressource geändert oder eine noch nicht existierende erzeugt. Der Server darf die übermittelten Daten ignorieren, ändern oder ergänzen. Der HTTP-Client kann nicht voraussetzen, dass genau die Daten, die er dem Server sendet, verwendet werden.

PUT ist wie GET und HEAD idempotent. Das heißt, dass ein einmaliges und ein mehrmaliges Aufrufen der Operation zum gleichen Ergebnis führen. Diese Bedingung ist sehr wichtig für eine verteilte Anwendung, denn falls beispielsweise ein mobiler Client mit eingeschränkter Netzwerkqualität sich nicht sicher ist, ob sein PUT-Request erfolgreich beim Server eintraf, kann er diesen wiederholen, ohne dass er sich um doppelte Einträge sorgen muss.

POST

Der primäre Zweck von POST ist das Anlegen einer neuen Ressource. Wie oben beschrieben wurde, erfüllt auch PUT diesen Zweck. Der Unterschied ist die Angabe der URI: Bei POST wird die Ressource unter einer vom Server gewählten URI angelegt. Bei PUT bestimmt der Client die URI mithilfe eines Location-Headers.

Weil POST keine semantischen Garantien erfüllen muss, wird es in der Regel zum Anstoßen von beliebigen Operationen eingesetzt, deren Aufruf den Zustand einer Ressource potenziell ändert. Dieser »Missbrauch« ist in der Praxis durchaus häufig anzutreffen. Beispielsweise nutzt JSON-RPC, ein Protokoll für Remote Procedure Calls, bevorzugt POST. Die Vorteile der anderen HTTP-Methoden werden dabei leider ignoriert.

»Lost Update«-Problem

Wenn mehrere Clients nahezu gleichzeitig eine Ressource ändern, könnte versehentlich ein Update verloren gehen. Um das zu verhindern, wird häufig in Kombination mit POST und anderen zustandsverändernden Methoden der If-Match-Header eingesetzt. Mit dem Header übertragene Entity Tags werden mit der aktuellen Repräsentation auf dem Server verglichen. Falls keines der Entity Tags übereinstimmt, wird die vom Client aufgerufene Methode nicht ausgeführt, sodass nicht unbeabsichtigt eine falsche Version der Ressource geändert wird. Falls die aufgerufene Ressource im Moment keine Repräsentation hat und ein Asterisk * übergeben wird, wird die Methode nicht ausgeführt[2].

DELETE

Wie der Name schon verrät, dient diese Methode zum Löschen von Ressourcen entsprechend der angegebenen URI. Das Löschen von etwas, das es nicht gibt, stellt kein Problem dar. Somit könnte man DELETE auch mehrfach aufrufen. Ein Client darf allerdings nicht voraussetzen, dass das Löschen oder Verschieben einer Ressource tatsächlich durchgeführt wurde, auch wenn der Statuscode darauf hindeutet. Ein erfolgreicher Aufruf sollte mit 200 »OK« bestätigt werden. Für akzeptierte, aber noch nicht durchgeführte Löschoperationen eignet sich 202 »Accepted«.

OPTIONS

Diese Methode liefert die möglichen Kommunikationsoptionen einer Ressource. Eine minimale Serverantwort wäre ein 200 »OK« mit einem Allow-Header, der die HTTP-Operationen auflistet, die für diese Ressource verwendet werden können.

```
HTTP/1.1 200 OK
ALLOW: HEAD, GET, PUT, OPTIONS
```

Der Nachrichtenrumpf der Antwort sollte weitere Informationen über diese Kommunikationsoptionen enthalten. Das Format dieser Optionen ist nicht festgelegt. OPTIONS könnte daher helfen, eine selbstbeschreibende API zu bauen, doch davon wird selten Gebrauch gemacht.

---

2. Falls Sie umgekehrt eine Methode nur für Ressourcen ohne Repräsentation ausführen wollen, können Sie den If-None-Match-Header mit Asterisk * nutzen. So können Sie beispielsweise die Ausführung einer duplizierten PUT-Nachricht verhindern.

PATCH

Viele Applikationen auf Basis von HTTP benötigen auch einen Mechanismus wie HTTP PATCH, mit dem nur ein Teil einer Ressource aktualisiert werden kann. Im Gegensatz dazu ersetzt HTTP PUT stets die vollständige Repräsentation der Ressource, die – sofern noch nicht bekannt – mit einem HTTP GET zuvor geladen werden muss. Wenn letztlich nur ein einzelnes Attribut geändert werden soll, ist der Overhead unverhältnismäßig hoch. Wenn ein Client mit einem PATCH nur die Daten überträgt, die geändert werden sollen, ist die Intention für den Server auch einfacher nachvollziehbar. PATCH ist weder sicher noch idempotent. Die Änderungen müssen vom Server atomar durchgeführt werden. Das bedeutet, dass der Server zu keinem Zeitpunkt unvollständig gepatchte Daten als Antwort auf ein GET-Request zurückgeben darf.

Bei einem Patch muss man sich beispielsweise Gedanken machen, wie mit partiellen Updates von Arrays umgegangen wird. Ein spezieller Medientyp wie JSON Patch kann genau für diese Aufgabe genutzt werden. Häufig verwendet man aber statt PATCH ein einfaches PUT auf eine Subressource, weil PUT geläufiger ist.

Zusammenfassung

Hier sind noch einmal die zuvor beschriebenen HTTP-Methoden im Überblick: PUT, POST, PATCH und DELETE führen zu Datenänderungen und sind daher nicht sicher. Entsprechend sollten Clients keine unbedachten Aufrufe mit diesen Methoden durchführen. Alle HTTP-Methoden, außer POST und PATCH, sind idempotent und führen auch bei wiederholten Aufrufen zum gleichen Ergebnis. Die Leseoperationen GET, HEAD und OPTIONS sind beides – sicher und idempotent.

HTTP-Methode	Sicher	Idempotent
GET	Ja	Ja
HEAD	Ja	Ja
PUT	Nein	Ja
POST	Nein	Nein
DELETE	Nein	Ja
OPTIONS	Ja	Ja
PATCH	Nein	Nein

Tab. 8–2
Eigenschaften der HTTP-Methoden

8.5 HATEOAS

Der Einsatz von Hypermedia ist eine wichtige Anforderung von REST. Auch Roy Fielding schrieb in seiner Doktorarbeit, dass REST-APIs »hypertext-driven« sein müssen. Diese Aussage bezieht sich auf HATEOAS, ein Konzept, das jedoch häufig von APIs missachtet wird, obwohl sie sich »RESTful« nennen. Die Abkürzung HATEOAS steht für »Hypermedia As The Engine Of Application State« und hat folgende Bedeutung:

- »Hypermedia« ist eine Verallgemeinerung des Hypertexts mit multimedialen Anteilen. Beziehungen zwischen Objekten werden durch Hypermedia Controls abgebildet.
- Mit »Engine« ist ein Zustandsautomat gemeint. Die Zustände und Zustandsübergänge der »Engine« beschreiben das Verhalten der »Application«.
- Im Kontext von REST kann man »Application« mit Ressource gleichsetzen.
- Mit »State« ist der Zustand der Ressource gemeint, deren Zustandsübergänge durch die »Engine« definiert werden.

Ein Zustandsautomat besteht aus Knoten, die durch Kanten miteinander verbunden sein können. Die Knoten entsprechen den Zuständen und die Kanten den Zustandsübergängen. Eine Kante führt demzufolge vom Ausgangszustand zum Folgezustand. Bei REST ist der Zustand einer Ressource vollständig in seiner Repräsentation enthalten und die Engine stellt die möglichen Zustände und deren Übergänge dar. Die Zustandsübergänge sind, da es sich um Hypermedia handelt, Links. Beispielsweise könnte eine Produktressource eine Operation zum Ändern der Produktbeschreibung anbieten. Wie später noch gezeigt wird, gibt es spezielle Hypermediaformate, die derartige Operationen genau beschreiben können. Auf diese Weise erfahren die Clients direkt von der Ressource, welche URI mit welcher HTTP-Methode und welchen Daten aufgerufen werden kann, um den Preis zu ändern.

Fundamentale Unterschiede zu Remote Procedure Calls

Der REST-Architekturstil definiert das Web als ein System verteilter Ressourcen, die über das Hypertext Transfer Protocol kommunizieren und durch Links miteinander verbunden sind. Aus diesem Grund ist HTTP ein passendes Fundament für eine REST-API. Im Zentrum einer REST-API stehen die Ressourcen, die durch Links erreichbar und miteinander vernetzt sind. Um den Zustand einer Ressource zu ändern, muss deren Repräsentation geändert und an die Ressource geschickt

werden. Die Ressource entscheidet dann, ob und wie sie ihren Zustand ändert. Dieser Ansatz unterscheidet sich grundlegend von Remote Procedure Calls. Hier übernimmt der Client die Kontrolle und ruft Prozeduren auf einem entfernten Server auf. Dazu muss der Client die Prozeduren des Servers kennen. Beide Seiten werden hierdurch aneinander gekoppelt. Auch SOAP-Webservices folgen genau diesem Ansatz. Im Gegensatz dazu behält bei REST die Ressource die Kontrolle über ihren Zustand. Nur sie kennt ihre internen Prozeduren. Sie gibt immer »nur« eine Repräsentation von sich heraus (daher auch der Begriff »representational state transfer«). Clients können eine solche Repräsentation beliebig verändern oder sogar löschen. Solange die Ressource nicht entscheidet, diese Änderung umzusetzen, bleibt die Ressource unverändert.

Ein zweiter wichtiger Unterschied zwischen REST und RPC sind die Links, über die die Ressourcen erreichbar sind. Weil die Ressource die Links ändern kann, hat sie die Macht, die API vollständig selbst zu bestimmen. Sie könnte beispielsweise bestimmte Links nur ausgewählten Clients anbieten. Diese beiden wesentlichen Unterschiede geben der REST-Ressource maximale Kontrolle und minimale Kopplung zu ihren Clients.

Die Produktressource im zuvor genannten Beispiel bietet einen Link zum Ändern der Produktbeschreibung. Falls diese Aktion nur von bestimmten Clients ausgeführt werden darf, braucht nicht jeder Client diesen Link zu erhalten[3]. Der Server kann selbst steuern, welcher Client den Link erhält.

Weil HTTP nur eine eingeschränkte Menge von Operationen (GET, PUT, POST etc.) unterstützt, kann das Ressourcenmodell nicht so flexibel wie ein internes Domänenmodell aufgebaut werden, sondern sollte diese Einschränkung berücksichtigen. Beim Entwurf einer REST-API muss deswegen die Anzahl der möglichen Operationen eingeschränkt werden. Die Logik der API basiert nicht auf Prozeduren, sondern auf Daten. Auf der anderen Seite ergibt sich durch die Verwendung eines bereits definierten, generischen Protokolls die Möglichkeit, Standardclients zu benutzen.

Dynamischer Workflow

Eine REST-API mit Hypermedia kann man sich wie eine Webseite vorstellen, deren API man mit einem Browser bedient. Ausgehend von der Startseite, deren Adresse man kennt, folgt man nur noch den angebo-

3. Sicherheitskritische APIs und deren Daten sind selbstverständlich auch durch Authentifizierung und Autorisierung abzusichern.

tenen Links. Welche Links angeboten werden, das entscheidet ganz allein die Ressource zur Laufzeit und nicht der Client auf Basis einer statisch definierten API. Der Workflow der Interaktion kann auf diese Weise sehr dynamisch durch Erzeugung der Links durch die Ressource gesteuert werden. Theoretisch hat die Ressource so die Kontrolle über die API und kann sogar URIs ändern, ohne dass Clients davon nachteilig betroffen sind, sofern diese nur die von der Ressource erhaltenen Links verwenden.

Viele vermeintliche REST-APIs ignorieren die Bedingungen von HATEOAS und verdienen nicht die Bezeichnung »RESTful«. Denn falls eine API keine Links anbietet, müssen Clients diese kennen und eventuell selbst zusammenbauen. Somit gibt es keinen dynamischen Workflow, sondern eine statische API, die fest in die Clients codiert werden muss. Falls der Workflow aufgrund neuer Anforderungen geändert wird, müssten ebenfalls die Clients programmatisch angepasst werden.

Affordance

Die wesentlichen Elemente eines Hypermediaentwurfs sind die Affordances. Für diesen Begriff gibt es leider keine einheitliche Übersetzung. Begriffe wie Aufforderungscharakter oder Angebotscharakter erscheinen noch am sinnvollsten. Der Begriff ist auch geläufig für UI-Designer und bezieht sich auf die Fähigkeit eines Objektes, sich selbst zu erklären. Eine grafische Oberfläche mit hoher Affordance ist im Idealfall intuitiv verständlich, sodass ein Benutzer sofort versteht, wie er die grafische Oberfläche bedienen kann.

Dieses Konzept lässt sich auch auf Hypermedia anwenden: In der Repräsentation kann ein Client Links zur Navigation und Transklusion erkennen. Außerdem bietet Hypermedia Elemente zur Zustandsänderung der Ressource. Falls keine derartigen Elemente vorhanden sind, ist die Ressource für den Client unveränderlich. Fall es jedoch Elemente zur Zustandsänderung gibt, sollten diese anzeigen, ob sie idempotent sind.

Hypertext Transfer Protocol

Hat HTTP alles, was man braucht, um HATEOAS umzusetzen? Um diese Frage beantworten zu können, muss man zunächst einmal klären, wie eine HTTP-Response aufgebaut ist: Sie besteht aus einem Header und einem Body. Im Header können Links angegeben werden, um einfache Hypermedia-Steuerelemente bereitzustellen. Diese Metainformationen gehören nicht zur eigentlichen Repräsentation der Ressource,

denn diese wird im Body angegeben. Ein Header eines Dienstes zur Verwaltung von Produkten könnte beispielsweise so aussehen:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: <http://api.example.com/>
       rel="self";
       type="text/html;charset=UTF-8";
       title="Servicebeschreibung und Homepage";
       verb="GET, HEAD",
       <http://api.example.com/products>
       rel="all";
       type="application/json;charset=UTF-8";
       title="Liste aller Produkte";
       verb="GET"
```

Der Link-Header besteht aus einer Liste von Linkeinträgen, die jeweils eine URL, Parameter und Extensions enthalten. Die Extensions können dazu verwendet werden, die erlaubten HTTP-Methoden eines Links zu beschreiben. Zu den Parametern gehört auch ein Relationstyp zur Angabe der Bedeutung des Links. Eine Auswahl der standardisierten Linkrelationen ist in der folgenden Tabelle aufgeführt:

Tab. 8–3 *Bedeutung standardisierter Linkrelationen*

Linkrelation	Bedeutung
all	Kennzeichnet einen Link, der zur Listenrepräsentation einer Ressource führt
new	Markiert einen Link zum Anlegen einer Ressource
next	Führt zum vorherigen Workflowschritt zurück
previous	Verweist auf den nächsten Schritt im Workflow
self	Kennzeichnet einen Link, der auf die aktuelle Ressourcenrepräsentation zeigt

Der Link-Header im obigen Beispiel bietet einen Self-Link, der die vorliegende Ressource identifiziert. Es handelt sich hierbei um eine Homepage der API, von der aus zu weiteren Ressourcen navigiert werden kann. Der Link kann sowohl mit HTTP GET als auch mit HTTP HEAD aufgerufen werden. HTTP HEAD kann benutzt werden, falls kein HTTP-Body benötigt wird.

Der zweite Link im obigen Beispiel führt den Client zu einer Liste mit Produkten. Die angegebene URL kann mit HTTP GET aufgerufen werden.

Nun zurück zur Frage, ob HATEOAS mit Standard-HTTP-Mitteln umgesetzt werden kann: Es ist tatsächlich möglich, eine HATEOAS-konforme API zu entwerfen, falls kein Hypermediaformat verwendet werden kann. Wie aber in Abschnitt 9.3 gezeigt wird, bieten spezielle

Hypermediaformate viele Vorteile. Beispielsweise können Links mit Templates und Links für idempotente Schreiboperationen definiert werden. Außerdem beziehen sich Link-Header immer nur auf ganze Ressourcen.

8.6 Zusammenfassung

In diesem Kapitel haben Sie REST als erfolgreichen Stil des Web kennengelernt. Einige wichtige Aussagen sind in der folgenden Liste noch einmal zusammengefasst:

- Um von der Funktionsweise des Web profitieren zu können, sollten Sie die Bedingungen von REST beim Entwurf einer API berücksichtigen.
- RESTful HTTP zeichnet sich durch Ressourcen mit eindeutigen URIs zur Identifikation, durch den Einsatz von Hypermedia, durch die Anwendung von HTTP-Standardmethoden unter Beachtung ihrer Semantik, durch unterschiedliche Ressourcenrepräsentationen und durch statuslose Kommunikation aus.
- Hypermedia ist eine zentrale Anforderung von REST: Der Workflow der API entspricht einem Zustandsautomaten, dessen Zustände die Ressourcen darstellen. Die möglichen Zustandsübergänge werden durch Links von den Ressourcen angeboten.
- Ressourcen können mit unterschiedlichen Medientypen repräsentiert werden.

Im nächsten Kapitel folgen praktische Techniken zum Entwurf von Web-APIs auf Basis von HTTP.