2 Qualitätsmerkmale

Nachdem Sie im vorherigen Kapitel die Prinzipien und den Zweck von APIs kennengelernt haben, geht es in diesem Kapitel weiter mit den allgemeinen Qualitätsmerkmalen. Diese Merkmale sind das Ziel der Best Practices und Design-Heuristiken in diesem Buch.

2.1 Allgemeine Qualitätsmerkmale

Um die Qualität eines Produktes oder einer Applikation bewerten zu können, gibt es viele Qualitätsmodelle, von denen sich insbesondere DIN/ISO 9126 in der Praxis durchgesetzt hat. Die darin definierten Qualitätsziele gelten für Software im Allgemeinen und damit auch für APIs. Ein Ziel ist beispielsweise die Richtigkeit der geforderten Funktionalität. Zweifellos ist das ein wichtiges Ziel, doch welche Ziele kann man für APIs besonders hervorheben?

Benutzbarkeit

- APIs sollen für andere Entwickler leicht verständlich, erlernbar und benutzbar sein. Gute Benutzbarkeit ist ein zentrales Ziel beim API-Design, deshalb finden Sie in diesem Kapitel weitere Informationen darüber.

Effizienz

- Insbesondere für mobile Applikationen ist geringer Akku-Verbrauch und geringes Online-Datenvolumen wichtig. Remote-APIs und eventuell dazugehörige Software Development Kits (SDKs), mit denen die Remote-APIs aufgerufen werden, sollten dies berücksichtigen. Auch Skalierbarkeit kann ein wichtiges Ziel sein, falls Sie beispielsweise davon ausgehen, dass Ihre API in Zukunft immer häufiger aufgerufen wird. Kapitel 14 bietet weitere Informationen zu diesem Thema.

Zuverlässigkeit

- Die Reife einer API-Implementierung hängt von der Versagenshäufigkeit durch Fehlerzustände ab. Interessant für API-Designer ist vor allem die Frage, wie man mit Fehlern umgehen soll. Informationen über Exception Handling und zur Fehlerbehandlung von Web-APIs finden Sie in Abschnitt 5.8 und 9.4.

2.2 Benutzbarkeit

Wann ist eine API gut benutzbar? Vermutlich kann diese Frage nur mit einer subjektiven Einschätzung beantwortet werden. Dennoch gibt es eine Reihe allgemein akzeptierter Eigenschaften. Weil aber diese Eigenschaften in der Praxis nie vollständig umgesetzt werden können, könnte man auch von Zielen sprechen:

- Konsistent
- Intuitiv verständlich
- Dokumentiert
- Einprägsam und leicht zu lernen
- Lesbaren Code fördernd
- Schwer falsch zu benutzen
- Minimal
- Stabil
- Einfach erweiterbar

Diese Eigenschaften werden in den folgenden Abschnitten vorgestellt.

2.2.1 Konsistent

Kohärentes Design mit der Handschrift eines Architekten

»Konsistenz« deckt sich weitestgehend mit »konzeptioneller Integrität«. Dieses Grundprinzip besagt, dass komplexe Systeme ein kohärentes Design mit der Handschrift eines Architekten haben sollten. Dieses Designprinzip stammt von Frederick Brooks, der bereits vor mehreren Jahrzehnten schrieb: »Konzeptionelle Geschlossenheit ist der Dreh- und Angelpunkt für die Qualität eines Produkts […]« [Brooks 2008]. Er meint damit, dass Entwurfsentscheidungen, wie beispielsweise Namensgebungen und die Verwendung von Mustern für ähnliche Aufgaben, im gesamten System durchgängig angewandt werden sollen. Das folgende Beispiel soll diese Aussage verdeutlichen. Zu sehen sind zwei Listen mit Funktionsnamen [Lacker 2013]:

```
str_repeat      strcmp
str_split       strlen
str_word_count  strrev
```

Die Liste auf der linken Seite beginnt mit einem Präfix »str«. Darauf folgen die Funktionsbezeichnungen, wobei die einzelnen Wörter durch Unterstriche voneinander getrennt sind. Die Funktionsnamen auf der rechten Seite sind ähnlich aufgebaut. Der Unterschied ist, dass man hier auf die Unterstriche verzichtet hat. Vermutlich haben Sie schon erkannt, dass es sich hierbei um die Bezeichnungen von Funktionen zur Bearbeitung von Zeichenketten handelt. Beide Namenskonventio-

nen sind in Ordnung. Es ist eine Frage des persönlichen Geschmacks, welche man bevorzugt.

Was ist das Problem? Das Problem ist, dass beide Namenskonventionen zur gleichen PHP-API gehören. Das bedeutet, dass sich Entwickler nicht nur die Namen der Funktionen, sondern auch ihre Namenskonvention merken müssen. Aus diesem Grund sollte eine API unbedingt die (nur eine) Handschrift eines Architekten[1] tragen.

Auch im Java Development Kit (JDK) lassen sich leicht Beispiele finden. Das Wort »Zip« wird im selben Package mal mit CamelCase und mal komplett in Großbuchstaben geschrieben:

```
java.util.zip.GZIPInputStream
java.util.zip.ZipOutputStream
```

Das Setzen des Textes eines Widgets ist nicht einheitlich im JDK gelöst. Mehrheitlich heißt die Methode setText, aber leider gibt es Abweichungen:

```
java.awt.TextField.setText();
java.awt.Label.setText();
javax.swing.AbstractButton.setText();
java.awt.Button.setLabel();
java.awt.Frame.setTitle();
```

2.2.2 Intuitiv verständlich

Die zweite wichtige Eigenschaft einer guten API ist intuitive Verständlichkeit. Eine intuitiv verständliche API ist in der Regel auch konsistent und verwendet einheitliche Namenskonventionen. Das bedeutet, dass gleiche Dinge die gleichen Namen haben. Und umgekehrt haben unterschiedliche Dinge auch unterschiedliche Namen. Dadurch ergibt sich eine gewisse Vorhersagbarkeit. Betrachten wir dazu ein weiteres Beispiel:

Ruby-Methoden mit Ausrufezeichen (!)

Ruby-Methoden, die mit einem Ausrufezeichen (!) enden, ändern das Objekt, auf dem sie aufgerufen wurden. Methoden ohne Ausrufezeichen am Namensende erzeugen hingegen eine neue Instanz und lassen das Objekt, auf dem sie aufgerufen wurden, unverändert.

```
my_string.capitalize
# Funktioniert wie capitalize, erzeugt aber keinen neuen String
my_string.capitalize!

my_string.reverse
# Funktioniert wie reverse, erzeugt aber keinen neuen String
my_string.reverse!
```

1. Selbstverständlich könnte es auch die Handschrift einer Architektin sein.

Nachdem Sie die Beispiele für capitalize! und reverse! gesehen haben, können Sie vermutlich das Namenspaar für »downcase« erraten.

Setter- und With-Mehoden

Für Java gibt es ebenfalls derartige Konventionen. Eine Konvention betrifft Setter-Methoden wie setName, setId oder setProperty. Setter-Methoden ändern das aufgerufene Objekt. Methoden wie withName, withId oder withProperty ändern das aufgerufene Objekt nicht, sondern erzeugen ein neues Objekt mit den angegebenen Werten. Das Präfix »with« wird beispielsweise von Joda-Time genutzt.

Methoden der Java-Collections

Ein anderes anschauliches Beispiel sind die Collections der Java-Standardbibliothek. Die starken Begriffe add, contains und remove wurden hier etabliert und da, wo es passt, wiederverwendet. Man findet diese Methoden einheitlich in den Interfaces List und Set. Für Map wurde leider ein anderer Name verwendet. Die Methode zum Hinzufügen von Elementen heißt dort put. Zugegeben, diese Methode hat andere Parameter und funktioniert nicht wie add von List, aber auch zwischen List und Set gibt es Unterschiede. Ein einheitliches add in allen drei Interfaces wäre vermutlich besser gewesen.

Die folgende Tabelle zeigt die Interfaces von List, Set und Map im Vergleich:

java.util.List	java.util.Set	java.util.Map
add	add	put
addAll	addAll	putAll
contains	contains	containsKey, containsValue
containsAll	containsAll	–
remove	remove	remove
removeAll	removeAll	–

Die Tabelle zeigt eine gewisse Symmetrie. Denn es gibt Methodenpaare wie add und remove, addAll und removeAll etc. Die Methode removeAll scheint bei Map zu fehlen, denn diese Methode wäre das Gegenstück zu putAll. Map bietet außerdem die Methode entrySet und keySet, aber nicht die Methode valueSet. Diese Methode heißt korrekterweise values, weil die Mengeneigenschaften in diesem Fall nicht erfüllt werden kann.

Es ist wichtig, starke Begriffe in einer API zu etablieren und diese einheitlich wiederzuverwenden. Beispielsweise sollten Sie nicht Synonyme wie »delete« und »remove« innerhalb einer API beliebig mischen. Entscheiden Sie sich für einen Begriff und bleiben Sie dann dabei. Verwenden Sie Begriffe, die den Benutzern der API geläufig sind. Das Wort »erase« wäre zum Beispiel zu ungewöhnlich. Ein Java-

Entwickler würde vermutlich für Dateioperationen nach »create« und »delete« als Erstes suchen. Bei Persistenzoperationen hingegen nach »insert« und »remove«.

Eine API ist intuitiv verständlich, wenn Entwickler den Clientcode der API ohne die Dokumentation lesen können. Das kann nur durch Vorwissen und sprechende Bezeichner funktionieren. Daher sollten Sie gezielt versuchen, Begriffe aus bekannten APIs wiederzuverwenden.

2.2.3 Dokumentiert

Eine API sollte möglichst einfach zu benutzen sein. Gute Dokumentation ist für dieses Ziel unverzichtbar. Neben Erklärungen für einzelne Klassen, Methoden und Parameter sollten auch Beispiele in der Dokumentation vorhanden sein. Entwickler können durch Beispiele schnell eine API lernen und benutzen. Im Idealfall findet ein Entwickler ein passendes Beispiel, das mit wenigen Änderungen direkt wiederverwendet werden kann. Die Beispiele der Dokumentation zeigen, wie die API korrekt verwendet werden soll.

Gute Dokumentation kann zum Erfolg einer Technologie beitragen. Das Spring Framework hat beispielsweise eine sehr gute Dokumentation mit vielen sinnvollen Beispielen und Erklärungen. Dies war sicherlich ein Grund für die hohe Akzeptanz des Frameworks.

2.2.4 Einprägsam und leicht zu lernen

Wie leicht oder schwer es ist, eine API zu lernen, hängt von vielen unterschiedlichen Faktoren ab. Eine konsistente, intuitiv verständliche und dokumentierte API ist sicherlich einfacher zu lernen als eine inkonsistente, unverständliche und undokumentierte. Die Anzahl der von einer API verwendeten Konzepte, die Wahl der Bezeichner und das individuelle Vorwissen der Benutzer haben ebenfalls großen Einfluss auf die Lernkurve.

APIs sind nur mit Mühe zu erlernen, wenn die Einstiegshürden sehr hoch gelegt werden. Dies ist dann der Fall, wenn viel Code für erste kleine Ergebnisse geschrieben werden muss. Nichts kann einen Benutzer mit Anfängerkenntnissen mehr einschüchtern. Das Webframework Vaadin bietet deswegen auf seiner Website ein Beispiel[2] mit geringer Einstiegshürde und »sichtbaren« Ergebnissen:

---

2. *https://vaadin.com/introduction#how-works*

```
public class MyUI extends UI {
     protected void init(VaadinRequest request) {
           TextField name = new TextField("Name");
           Button greetButton = new Button("Greet");
           greetButton.addClickListener(
               e -> Notification.show("Hi " + name.getValue())
           );
          setContent(new VerticalLayout(name, greetButton));
     }
}
```

Das Beispiel zeigt die Verwendung von Widgets – eine Besonderheit für Webframeworks. Dieses Beispiel hat den Vorteil, dass mit nur etwa 10 Zeilen Code ein erstes sichtbares Ergebnis entsteht. Das Beispiel kann man für weitere Experimente nutzen, um das Framework auszuprobieren.

2.2.5 Lesbaren Code fördernd

APIs haben enormen Einfluss auf die Lesbarkeit des Clientcodes. Schauen wir uns dazu folgendes Beispiel an:

```
assertTrue(car.getExtras().contains(airconditioning));
assertEquals(2, car.getExtras().size());
```

Das Beispiel ist ein Auszug aus einem Unit Test. Die beiden Assertions prüfen, ob das Fahrzeug car eine Klimaanlage und insgesamt zwei Extras hat. Alternativ könnte der Unit Test auch mit dem FEST-Assert-Framework geschrieben werden:

```
assertThat(car.getExtras())
    .hasSize(2)
    .contains(airconditioning);
```

Dank des Fluent Interface, dessen Methodenketten zur Validierung des Testergebnisses stärker an eine natürliche Sprache angelehnt sind, ist der Code des zweiten Beispiels etwas verständlicher. Ein Fluent Interface ist eine Domain Specific Language (DSL), die durch die Anpassung an die Anforderungen ihrer Domäne viel ausdrucksstärker als eine universelle Programmiersprache ist. In Abschnitt 6.1 finden Sie weitere Informationen zu diesem Thema.

Bessere Lesbarkeit und Wartbarkeit von Unit Tests waren die Entwurfsziele des FEST-Assert-Frameworks. In diesem Zusammenhang könnte man noch viele andere Bibliotheken mit gleichem Zweck nennen: Das Spock Famework beispielsweise bietet eine kleine DSL zur übersichtlichen Strukturierung von Tests.

Ein Beispiel aus einem ganz anderen Aufgabengebiet ist die JPA Criteria API. Diese API dient zur Konstruktion von typsicheren Daten-

bankabfragen. Mit dem folgenden Java-Code wird eine Query gebaut und ausgeführt, um alle Order-Objekte mit mehr als einer Position zu selektieren:

```
EntityManager em = ...;
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Order> cq = builder
   .createQuery(Order.class);
Root<Order> order = cq.from(Order.class);
order.join(Order_.positions);
cq.groupBy(order.get(Order_.id)).having(
   builder.gt(builder.count(order), 1));
TypedQuery<Order> query = em.createQuery(cq);
List<Order> result = query.getResultList();
```

Übersichtlicher wird die Abfrage mit QueryDSL. Diese Bibliothek bietet ein Fluent Interface, mit dem verständliche Pfadausdrücke formuliert werden können.

```
EntityManager em = ...;
QOrder order = QOrder.order;
JPQLQuery query = new JPAQuery(em);
List<Order> list = query.from(order)
                        .where(order.positions.size().gt(1))
                        .list(order).getResults();
```

Entwickler verbringen mehr Zeit mit dem Lesen als mit dem Schreiben von Quellcode. Daher kann deren Produktivität durch gut lesbaren Quellcode bzw. einer leicht verständlichen API verbessert werden. Wie können APIs zu lesbarem Code führen?

- Gute Namenskonventionen sind wichtig, denn sie unterstützen das Lesen und Erfassen des Quellcodes. Gut lesbarer Code enthält auch weniger Fehler, denn Fehler fallen dann schneller auf.
- Die zuvor beschriebenen Eigenschaften Konsistenz und intuitive Verständlichkeit haben ebenfalls einen großen Einfluss auf die Lesbarkeit des Clientcodes.
- Auch ein einheitliches Abstraktionsniveau verbessert die Lesbarkeit von Code. Das bedeutet, dass eine API beispielsweise nicht Persistenzfunktionen mit Geschäftslogik mischen sollte. Das sind Aufgaben unterschiedlicher Abstraktionsniveaus. Wenn diese vermischt werden, entsteht unnötig komplexer Clientcode. Die gewählten Abstraktionen der API sollten passend für die zukünftigen Benutzer ausgewählt werden.
- APIs sollten Hilfsmethoden bieten, sodass der Clientcode kurz und verständlich bleibt. Ein Client sollte nichts tun müssen, was ihm die API abnehmen kann.

2.2.6 Schwer falsch zu benutzen

Eine API sollte nicht nur einfach zu benutzen, sie sollte sogar schwer falsch zu benutzen sein. Daher sollte man nicht offensichtliche Seiteneffekte vermeiden und Fehler zeitnah mit hilfreichen Fehlermeldungen anzeigen. Benutzer sollten nicht gezwungen sein, die Methoden einer API in einer fest definierten Reihenfolge aufzurufen.

Unerwartetes Verhalten

Die ursprüngliche Datums- und Zeit-API von Java sieht auf den ersten Blick einfach und intuitiv aus. Doch schon bei einfachen Beispielen stolpert man über ein Verhalten, das man vermutlich nicht erwartet. Was das bedeutet, wollen wir uns an einem Beispiel anschauen:

Die ursprüngliche Datums- und Zeit-API von Java lädt geradezu dazu ein, Fehler zu machen. Den 20. Januar 1983 würde man vermutlich so definieren wollen:

```
Date date = new Date(1983, 1, 20);
```

Leider enthält diese Codezeile gleich zwei Fehler. Denn die Zeitrechnung dieser API beginnt unerwarteterweise im Jahre 1900. Außerdem sind die Monate beginnend mit 0 durchnummeriert. Die Tage werden beginnend mit 1 angegeben. Deswegen muss der 20. Januar 1983 folgendermaßen erzeugt werden:

```
int year = 1983 – 1900;
int month = 1 - 1;
Date date = new Date(year, month, 20);
```

Im nächsten Schritt geben wir zusätzlich noch die Uhrzeit 10:17 mit der Zeitzone von Bukarest an. Die Uhrzeit soll schließlich in einen formatierten String umgewandelt werden. Weil die Klasse Date keine Zeitzonen unterstützt, müssen wir ein Calendar-Objekt erzeugen. Die erwartete Ausgabe ist »20.01.1983 10:17 +0200«.

```
Date date = new Date(year, month, 20, 10, 17);
TimeZone zone = TimeZone.getInstance("Europe/Bucharest");
Calendar cal = new GregorianCalendar(date, zone);
DateFormat fm = new SimpleDateFormat("dd.MM.yyyy HH:mm Z");
String str = fm.format(cal);
```

Auch hier verstecken sich mehrere Fehler: Der Konstruktor der Klasse GregorianCalendar akzeptiert eine Zeitzone, aber kein Date-Objekt. Der Calendar kann nicht von SimpleDateFormat formatiert werden. Auch SimpleDateFormat muss die Zeitzone übergeben werden. Durch Angabe der Zeitzone wird die Uhrzeit verändert. Der korrigierte Clientcode sieht so aus:

```
int year = 1983 - 1900;
int month = 1 - 1;
// weil 1 Stunde Zeitunterschied zwischen Berlin und Bukarest
int hour = 10 - 1;
Date date = new Date(year, month, 20, hour, 17);
TimeZone zone = TimeZone.getInstance("Europe/Bucharest");
Calendar cal = new GregorianCalendar(zone);
cal.setTime(date);
DateFormat fm = new SimpleDateFormat("dd.MM.yyyy HH:mm Z");
fm.setTimeZone(zone);
Date calDate = cal.getTime();
String str = fm.format(calDate);
```

Aufgrund dieser Fallstricke entstand in der Java-Community die Bibliothek Joda-Time. Der Clientcode könnte folgendermaßen aussehen:

```
DateTime dt = new DateTime(1983, 1, 20, 10, 17,
        DateTimeZone.forID("Europe/Bucharest"));
DateTimeFormatter formatter
        = DateTimeFormat.forPattern("dd.MM.yyyy HH:mm Z");
String str = dt.toString(formatter);
```

Ein anderes nicht unbedingt intuitives Feature ist die Möglichkeit, mehr als 60 Sekunden, mehr als 24 Stunden usw. bei der Erzeugung eines Date-Objektes anzugeben. Statt einer Fehlermeldung wird der Überhang korrekt berechnet. Durch eine Angabe von beispielsweise 25 Stunden wird der nächste Tag 1 Uhr ausgewählt. Dieses Verhalten ist nicht offensichtlich und könnte deswegen zu Fehlern führen.

2.2.7 Minimal

Im Zweifel weglassen!

Eine API sollte prinzipiell so klein wie möglich sein, weil einmal hinzugefügte Elemente nachträglich nicht mehr entfernt werden können. Außerdem sind größere APIs auch komplexer. Dies hat Auswirkungen auf Verständlichkeit und Wartbarkeit der API. Ein ganz anderer Punkt ist der Implementierungsaufwand: Je größer die API, desto aufwendiger ihre Implementierung. Deswegen sollten beispielsweise zusätzliche Hilfsmethoden nur mit Bedacht hinzugefügt werden. Andererseits können Hilfsmethoden sehr nützlich sein. Überhaupt sollte ein Client nichts tun müssen, was eine API übernehmen kann.

Daher braucht man einen Kompromiss, wie ihn die Entwickler der Java-Collection-API gefunden haben: Mit den Methoden addAll und removeAll im Interface java.util.List können mit einem Aufruf mehrere Objekte zu einer Liste hinzugefügt bzw. entfernt werden. Diese Methoden sind optional, weil man Objekte auch einzeln mit add und remove hinzufügen bzw. entfernen kann. Trotzdem ist das Vorhandensein dieser Hilfsmethoden im Interface java.util.List nachvollziehbar

und akzeptabel. Diese Hilfsmethoden werden sehr häufig verwendet und passen gut zum Rest des Interface. Andere Hilfsmethoden wie beispielsweise removeAllEven oder removeAllOdd, die alle Objekte mit gerader bzw. ungerader Positionsnummer aus einer Liste entfernen, wären für nur wenige spezielle Anwendungsfälle hilfreich und gehören deswegen nicht in die API.

Die Ruby-Standardbibliothek hat diverse Methoden mehrfach, weil man so im Clientcode besser ausdrücken kann, was man tut. Die Anzahl der Elemente eines Arrays kann z.B. mit length, count und size abgefragt werden. Das muss man nicht ermöglichen, aber es ist ein guter Stil, wenn man ihn konsistent anwendet.

Weniger ist manchmal mehr

Schweizer Messer sind bekannt für ihre zahlreichen Werkzeuge. Neben einer Klinge bieten sie z.B. eine Holzsäge, einen Korkenzieher, eine Schere, eine Metallfeile oder eine Pinzette. Manche dieser Werkzeuge werden kaum oder vielleicht nie benutzt. Ein gewöhnlicher Schraubenzieher mit einem vergleichsweise einfachen Design ist ebenfalls vielseitig einsetzbar: Man kann beispielsweise eine Farbdose mit ihm öffnen, falls der Deckel klemmt. Man kann mit ihm die Farbe umrühren, ein Loch in etwas machen, etwas hinter dem Schrank hervorholen, das man mit der Hand nicht erreichen kann, und man kann sogar Schrauben festdrehen.

Auch eine kleine einfache API kann vielseitig einsetzbar sein. Es muss nicht für jeden Sonderfall eine spezielle Funktion, die am Ende kaum jemand nutzen wird, eingebaut werden. Nichtsdestotrotz sind Schweizer Messer sehr nützlich.

2.2.8 Stabil

Stabilität ist eine wichtige Eigenschaft von APIs. Angenommen Sie entwickeln einen Tarifrechner. Ihr Produkt wird ein großer Erfolg und soll in mehrere Kundensysteme integriert werden. Die Integrationen werden von unterschiedlichen Teams durchgeführt und sind relativ teuer, weil Altsysteme nur mit großem Aufwand angepasst werden können. Dann gibt es eine neue Anforderung aus der Fachabteilung und die komplexen Berechnungsregeln des Tarifrechners müssen erweitert werden. Falls sich nun die Schnittstelle oder das bisherige Verhalten dieser Schnittstelle ändern würde, gäbe es Probleme bei der Integration in die Altsysteme. Deswegen muss bei jeder Änderung geprüft werden, ob diese negative Auswirkungen auf bestehende Benutzer hat und wie diese gegebenenfalls kommuniziert werden können. In Kapitel 7 werden wir uns anschauen, welche Änderungen kompatibel sind. Falls

Änderungen nicht kompatibel sind, ist gegebenenfalls eine neue Version zu nutzen. Stabilität ist auch bei Einführung einer neuen Version wichtig, wenn Sie die Migration für existierende Clients möglichst einfach machen wollen.

2.2.9 Einfach erweiterbar

Eine weitere Eigenschaft von APIs ist Änderbarkeit, ein zentrales Qualitätsmerkmal für Softwareprodukte. Man versteht darunter den erforderlichen Aufwand zur Durchführung vorgegebener Änderungen für Korrekturen, Verbesserungen oder Anpassungen an neue Anforderungen. Diese Eigenschaft ist kein Widerspruch zur zuvor genannten Stabilität, denn gemeint ist Folgendes:

- Bei der Erweiterung einer API sollte der Änderungsaufwand für existierende Clients berücksichtigt werden. Im nächsten Abschnitt wird aus diesem Grund die Metrik Konnaszenz vorgestellt.
- Im Idealfall ist die veränderte API kompatibel und der Clientcode muss überhaupt nicht angepasst werden.

Eine Java-API kann beispielsweise durch Vererbung erweitert werden:

- API-Benutzer können durch Vererbung das Verhalten eines Frameworks anpassen oder ändern.
- API-Entwickler können eine neue Subklasse hinzufügen, um auf kompatible Art und Weise neue Funktionen umzusetzen. Die neue Subklasse wird womöglich in einer Factory-Methode erzeugt, sodass API-Benutzer dies nicht bemerken.

Auch für Web-APIs ist Änderbarkeit ein wichtiges Qualitätsmerkmal. Flexible Datenformate wie XML und JSON können genutzt werden, um kompatible Erweiterungen umzusetzen.

2.3 Konnaszenz

Für leicht änderbaren Code ist noch keine Zauberformel erfunden worden, aber mit Konnaszenz (engl. Connascence) steht ein gutes Modell zur Verfügung, mit dem wir zumindest über die Änderbarkeit von APIs sprechen können.

Was ist Konnaszenz?

Konnaszenz berücksichtigt verschiedene Typen von Kopplung sowie deren Häufigkeit und Lokalität für Aussagen über die Änderbarkeit von Code (The connascence.io website, 2018). Zwei Komponenten A und B

gelten im Sinne von Konnaszenz als gekoppelt, falls eine Änderung an A eine Änderung an B erfordert, um die Korrektheit des Gesamtsystems zu gewährleisten. Eine starke Form von Konnaszenz zwischen A und B ist nur akzeptabel, wenn beide Komponenten eng beieinander liegen, weil sie zum Beispiel Teil derselben Codebasis sind. Umgekehrt ist eine starke Form von Konnaszenz zwischen Komponenten unterschiedlicher Systeme inakzeptabel. Es gilt: Je stärker die Konnaszenz zwischen A und B, desto schwieriger und aufwendiger sind deren Änderungen. Konnaszenz wird mit den Dimensionen Stärke, Häufigkeit und Lokalität beschrieben. Dies kann beim API-Design genutzt werden, um die Verbindung zwischen API und Client zu analysieren:

Stärke

- Stärkere Formen von Konnaszenz sind schwerer zu finden oder zu ändern als leichtere Formen. Beispielsweise sind Namensänderungen leicht zu entdecken und durchzuführen. Im Vergleich dazu ist die Anpassung eines Algorithmus, der zwischen API und Client abgestimmt werden muss, schwierig.

Häufigkeit

- Die Änderung einer API, die sehr viele Clients hat, ist höchstwahrscheinlich eine größere Herausforderung als die Änderung einer API mit wenigen Clients. Die Änderbarkeit von öffentlichen APIs mit vielen Clients ist daher stark eingeschränkt.

Lokalität

- Je weiter API und Client entfernt sind, desto schwieriger sind Änderungen. Wenn API und Client zur selben Codebasis gehören, ist die Änderung vergleichsweise einfach. Änderungen innerhalb des Zuständigkeitsbereichs eines Entwicklungsteams sind ebenfalls akzeptabel. Aber Änderungen über Organisationsgrenzen hinweg sind deutlich komplexer, weil die Organisationen wahrscheinlich verschiedene Ziele verfolgen, unterschiedliche Releasezyklen haben und die Kommunikation insgesamt schwieriger oder aufwendiger ist als innerhalb eines Teams.

Statische Konnaszenz

Es gibt verschiedene Formen von Konnaszenz, die man in statisch und dynamisch einteilen kann. Statische Formen kann man durch Lesen des Codes finden und analysieren. Die folgende Liste ist nach aufsteigender Stärke sortiert:

Namenkonnaszenz

- Die schwächste Form statischer Konnaszenz ist Namenskonnaszenz. In diesem Fall müssen sich mehrere Komponenten auf den Namen eines Elements einigen. Falls zum Beispiel ein Name im JSON-Payload einer API geändert wird, müssen auch die von diesem Format abhängigen Clients angepasst werden.

- Mehrere Komponenten müssen sich auf den Typ eines Elements einigen. Die Änderung eines Integers im JSON-Format einer API in einen String ist eine inkompatible Änderung mit mittelschwacher Konnaszenz. *Typkonnaszenz*
- Mehrere Komponenten müssen sich auf die Bedeutung bestimmter Werte einigen. Wenn eine API in ein Feld *preis* statt des Nettopreises nun den Bruttopreis schreibt, müssen die Clients angepasst werden. *Bedeutungskonnaszenz*
- Mehrere Komponenten müssen sich auf die Reihenfolge bestimmter Werte einigen. Die Reihenfolge von Parametern für einen API-Aufruf ist ein typisches Beispiel für Positionskonnaszenz. *Positionskonnaszenz*
- Bei der stärksten Form statischer Konnaszenz müssen sich mehrere Komponenten auf einen bestimmten Algorithmus einigen. Verbindungen zwischen API und Clients dieser Stärke können nur mit großem Aufwand geändert werden, daher sollte man beim Entwurf einer API auf Algorithmuskonnaszenz achten und versuchen, diese zu minimieren. Denken Sie beispielsweise an IBANs, die einem bestimmten Format folgen. Der Algorithmus zur Validierung von IBANs wird von unzähligen Systemen genutzt. Eine Änderung dieses Algorithmus wäre äußerst schwierig. *Algorithmuskonnaszenz*

Dynamische Konnaszenz

Wie bereits erwähnt wurde, gibt es neben den statischen auch dynamische Formen von Konnaszenz, die nachfolgend beschrieben werden:

- Die Methoden einer API können nur in spezieller Reihenfolge aufgerufen werden. Häufig gibt es einen offensichtlichen fachlich-logischen Grund, warum API-Methoden nicht in beliebiger Reihenfolge benutzt werden können. Nehmen wir zum Beispiel einen Webservice einer Shopping-Anwendung zum Verwalten von elektronischen Einkaufswagen. Man kann beispielsweise nicht mit einer Operation CartAdd einen Artikel in einen Einkaufskorb legen, wenn nicht zuvor mit einer Operation CartCreate ein entsprechendes Objekt (Einkaufskorb) mit dem Webservice erzeugt wurde. *Ausführungskonnaszenz*
- Diese Form bezieht sich auf die zeitliche Koordinierung zwischen API und Client. Fachliche Transaktionen werden beispielsweise für den Check-out-Prozess der Shopping-Anwendung eingesetzt. Nach Ablauf einer Frist wird die Transaktion automatisch vom Webserver gelöscht, es sei denn, die Transaktion wird vorher vom Client erfolgreich beendet oder abgebrochen. *Zeitliche Konnaszenz*

Wertekonnaszenz

- Diese Form von Konnaszenz liegt vor, wenn sich mehrere Werte zusammen ändern müssen, wenn beispielsweise Client und Server sich auf bestimmte Zahlen einigen, um den Zustand von Bestellungen anzugeben. Eine Bestellung in Bearbeitung hat dann zum Beispiel den Wert 2 und eine abgeschlossene Bestellung den Wert 3.

Identitätskonnaszenz

- In diesem Fall müssen mehrere Komponenten dieselbe Entität referenzieren. Angenommen die zuvor erwähnte Shopping-Anwendung besteht aus mehreren getrennten Webservices für die Verwaltung der elektronischen Einkaufswagen, für den Check-out-Prozess und für die Artikelsuche. Identitätskonnaszenz liegt vor, wenn die Webservices für Check-out und Artikelsuche denselben Einkaufswagen referenzieren müssen.

Konnaszenz gibt Ihnen das notwendige Vokabular, um die Änderbarkeit von APIs zu untersuchen und die vielfältige Kopplung zwischen Client und API zu benennen.

2.4 Zusammenfassung

In diesem Kapitel haben Sie die Qualitätsmerkmale bzw. Qualitätsziele kennengelernt. Diese sind hier zusammengefasst:

- APIs müssen vollständig und korrekt sein.
- APIs sollten konsistent, intuitiv verständlich, dokumentiert, minimal, stabil, erweiterbar und leicht zu lernen sein. Sie sollten es Benutzern leicht machen, lesbaren Code zu schreiben. Es sollte schwer sein, sie falsch zu benutzen.
- Die Änderbarkeit von APIs kann mit der Metrik Konnaszenz systematisch analysiert werden.

Im folgenden Kapitel werden Sie erfahren, wie APIs auf Basis von Use Cases und Beispielen entsprechend zuvor identifizierter Anforderungen iterativ mit Feedbackschleifen entworfen werden können.

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 der 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 erwarten 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.

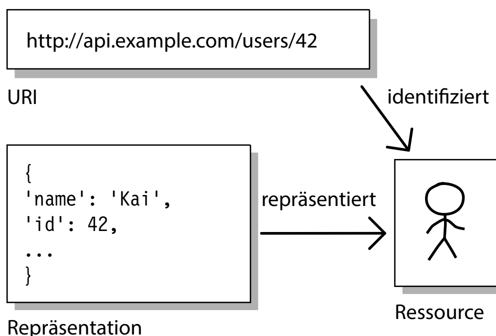

Abb. 8–1
Zusammenhang zwischen URI, Repräsentation und abstrakter Ressource

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 neue Ressource unter einer vom Server gewählten URI angelegt. Diese URI wird dem HTTP-Client im Location-Header der Antwort mitgeteilt. Bei PUT bestimmt der Client die URI der neuen Ressource. Es handelt sich schlicht um die URI des Aufrufs.

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, bevor-

zugt 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.

Tab. 8–2
Eigenschaften der HTTP-Methoden

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

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.

Unterschiede zu Remote Procedure Calls

Ein Remote Procedure Call (RPC) ist ein Mechanismus, der es einem Client ermöglicht, eine Prozedur in einem anderen Prozess aufzurufen oder eine Nachricht mit diesem auszutauschen. Typischerweise wird auf dem Client ein Vertreter-Stub, mit dem der Aufruf lokal erscheint, eingesetzt. Für den Nachrichtenaustausch kann HTTP verwendet werden, aber RPC ist nicht an HTTP gebunden.

Wie unterscheidet sich RPC von REST? Der Vergleich ist schwierig, denn eine REST-API kann auf Basis einer RPC-Implementierung umgesetzt werden. REST ist ein Architekturstil mit Bedingungen wie zustandsloser und cachbarer Client-Server-Kommunikation. Der Server bietet den Clients Repräsentationen seiner Ressourcen und bestimmt, ob und wie diese manipuliert werden können. Der Vertrag zwischen Client und Server ist sehr klein, denn der Client muss nur die URI der Homepage-Ressource, die er mit HTTP GET aufrufen kann, kennen. Alle weiteren Aktionen braucht der Client nicht zu kennen, denn sie werden ihm vom Server mitgeteilt.

Die Produktressource im zuvor genannten Beispiel bietet einen Link zum Ändern der Produktbeschreibung. Falls ein Client die Produktbeschreibung nicht ändern darf, weil bestimmte Voraussetzungen nicht erfüllt werden, gibt der Server diesen Link nicht an. Der Server kann selbst steuern, welcher Client den Link erhält. Um diese Hypermedia-Vorteile nutzen zu können, brauchen die Clients für REST-APIs mehr Protokollwissen als für APIs ohne Hypermedia. Dennoch werden die Clients nur lose an die REST-APIs gekoppelt, weil die Verteilung der Geschäftslogik und des Fachwissens entscheidend ist. Durch den Einsatz von Hypermedia braucht der Client weniger Fachwissen und die Verteilung der Geschäftslogik wird minimiert.

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 angebotenen 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:

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

Tab. 8–3
Bedeutung standardisierter Linkrelationen

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.