3., aktualisierte und erweiterte Auflage

Tilkov · Eigenbrodt · Schreier · Wolf

REST und HTTP

Entwicklung und Integration nach dem Architekturstil des Web

dpunkt.verlag

Inhaltsverzeichnis

4 Ressourcen

> *»There are only two hard things in Computer Science: cache invalidation and naming things« – Phil Karlton*

Das zentrale Konzept von REST sind Ressourcen. Im ersten Kapitel haben wir schon erwähnt, dass Ressourcen ein relativ abstraktes, sehr generisches Konzept sind, für das man genauso gut die Begriffe »Ding« oder »Objekt« hätte verwenden können. Eine Ressource zeichnet sich durch zwei wesentliche Eigenschaften aus: Sie ist identifizierbar und hat eine oder mehrere Repräsentationen, die sie ihrer Außenwelt zur Verfügung stellt und über die sie bearbeitet wird. In diesem Kapitel werden wir uns mit Ressourcen, deren Identifikation und dem Repräsentationskonzept beschäftigen.

4.1 Ressourcen und Repräsentationen

Wir sehen Ressourcen nie selbst – wir sehen nur ihre Repräsentationen. Repräsentationen sind Darstellungen einer Ressource in einem definierten Format, und jede Ressource kann mehrere Repräsentationen haben. Für eine Ressource »Person« zum Beispiel könnten eine einfache textuelle Repräsentation, eine HTML-Darstellung und ein Bild im JPEG-Format existieren. Keine davon ist die »richtige« oder auch nur kanonische Darstellungsform, alle sind gleichermaßen gültig. Daher der philosophisch leicht fragwürdige Bezug zu Platons Höhlengleichnis: Die darin beschriebenen Gefangenen sehen nie die Dinge der wirklichen Welt, sondern nur die Schatten, die diese werfen. Analog dazu sehen wir nie die Ressourcen, sondern immer nur deren Repräsentationen. Tatsächlich können wir die Ressource sogar darüber definieren: als eine durch eine gemeinsame ID zusammengehaltene Menge von Repräsentationen[1].

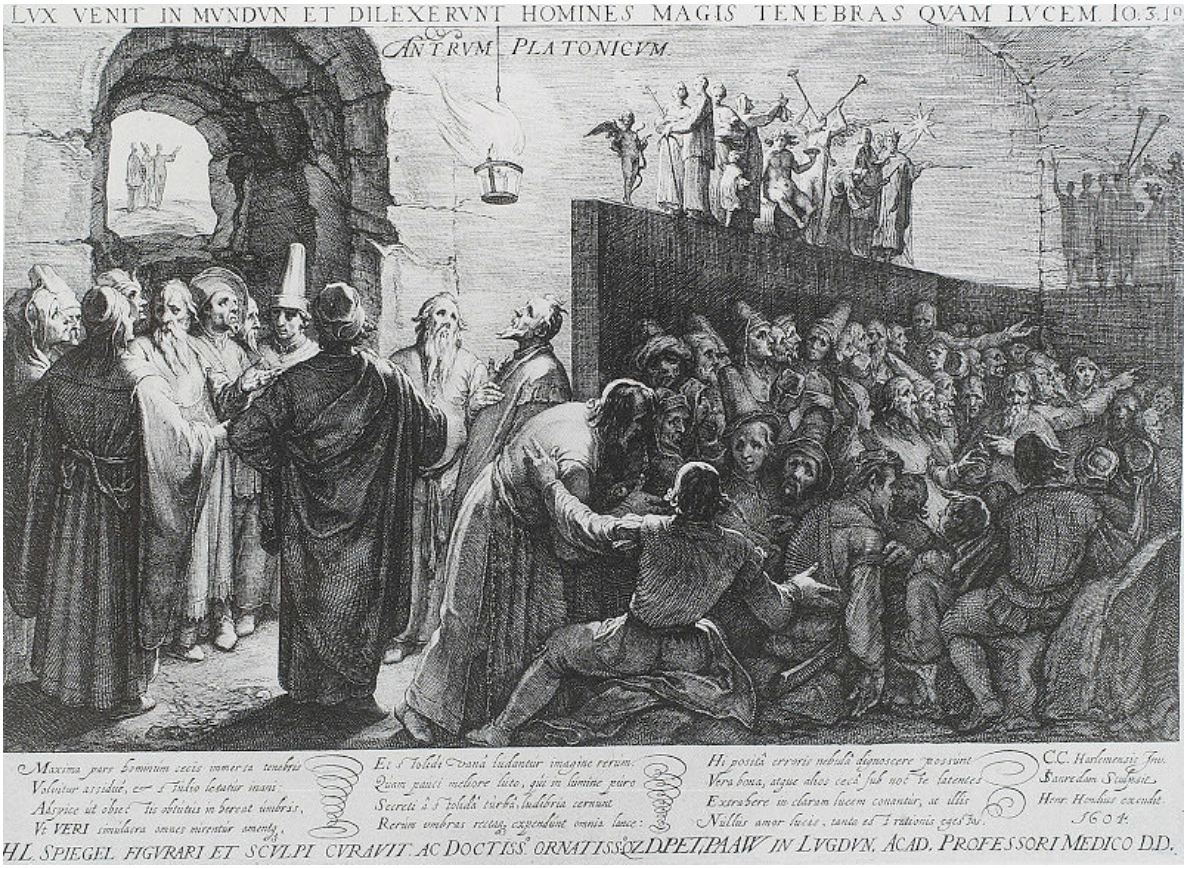


Abb. 4–1 *Höhlengleichnis des Platon, Stich von Jan Saenredam (1565-1607)*

Häufig ist die Grenze zwischen einer Ressource mit mehreren Repräsentationen auf der einen und separaten Ressourcen auf der anderen Seite fließend: Sind die Visitenkarte einer Person und ihr Bild wirklich zwei unterschiedliche Repräsentationen derselben Ressource? Oder handelt es sich um zwei unterschiedliche Ressourcen? Für beides kann man gute Argumente finden. In diesem spezifischen Fall tendieren wir zu separaten Ressourcen: Ein Bild enthält andere Informationen als die Visitenkarte, und vielleicht möchten Sie das Bild in einem bestimmten Kontext mit Zusatzinformationen verknüpfen, nicht aber die Details aus der Visitenkarte (oder andersherum). Schließlich können sowohl Bild als auch Visitenkarte jeweils in unterschiedlichen Formaten vorliegen: Ein weiterer Grund, die Information auf mehr als eine Ressource abzubilden.

Nicht umsonst werden Architekturen, die sich am REST-Stil orientieren, auch als ROA *(Resource-Oriented Architecture)* bezeichnet: Ressourcen stehen in der Regel im Mittelpunkt Ihres Entwurfs. Dem richtigen Design Ihrer Ressourcen kommt daher eine besonders hohe Bedeutung zu.

4.2 Ressourcendesign

Wenn Sie die Ressourcen für Ihren Anwendungsfall entwerfen, beschreiben Sie damit deren Schnittstelle zur Außenwelt. Wie bei jedem Schnittstellenentwurf sollten Sie sich Gedanken darüber machen, welche Informationen Sie veröffentlichen wollen und welche Implementationsdetails sind.

Zur besseren Strukturierung unterscheiden wir im Folgenden unterschiedliche Ressourcenkategorien[2]: Primärressourcen, Listenressourcen, Filter, Projektionen, Aggregationen, Aktivitäten und Konzepte.

4.2.1 Primärressourcen

Wenn Sie Ihre fachliche Domäne betrachten – das Umfeld, für das Sie eine Anwendung entwerfen –, dann werden Sie mit großer Wahrscheinlichkeit relativ einfach eine ganze Reihe von Ressourcen identifizieren können. Das Gleiche gilt, wenn Sie eine bestehende Anwendung mit einer Schnittstelle nach dem REST-Prinzip versehen möchten. In aller Regel sind die fachlichen Kernkonzepte die unmittelbar sinnvollen Kandidaten. Beispiele dafür wären: jeder einzelne Kunde und seine Adresse in einem CRM-System, jedes Konto, jede Mahnung und jeder Zahlungseingang in einer Anwendung zur Unterstützung des Mahnwesens oder Benutzer und Konversation in einem Chat-Server. Wir werden diese Art von Ressourcen im Folgenden »Primärressourcen« nennen; gemeint sind damit diejenigen Konzepte, die sich in der Regel bei einem klassischen Anwendungsentwurf sehr früh als Kandidaten für persistente Entitäten herauskristallisieren.

Man kann sich die Primärressourcen, die wir als Beispiele genannt haben, besser in Verbindung mit ihrer hypothetischen URI vorstellen[3]:

- `http://example.com/customers/1234`
- `http://example.com/accounts/DE731288112`
- `http://example.com/reminders/2761723`

Für den Nutzer Ihrer Ressourcen, sei es ein Webbrowser oder eine andere Anwendung, ist die Implementierung, die hinter den Ressourcen steht, vollständig transparent. Auch die Tatsache, dass die Ressourcen als identifizierbares Konzept Bestandteil der Schnittstelle sind, lässt keinen Rückschluss auf die Implementierung zu: Sie könnten sich als Zeilen in mehreren Datenbanktabellen mit komplexen Beziehungen, als Objekte im Hauptspeicher eines Anwendungssystems oder als Dateien auf einer Festplatte

manifestieren. Insbesondere ist die Granularität – anders gesagt, der Schnitt – der Ressourcen in aller Regel nicht deckungsgleich mit einer Implementierung. Verwechseln Sie Primärressourcen also nicht mit Datensätzen in Ihrer Datenbank!

Bestes Beispiel für eine Primärressource in unserem OrderManager-Beispiel ist die Bestellung selbst. Wir können uns mit GET über den aktuellen Status informieren oder mit PUT eine Aktualisierung vornehmen.

4.2.2 Subressourcen

Ressourcen, die Teil einer anderen Ressource sind, bezeichnen wir als Subressourcen. Beispiele können die einzelnen Bestellungen in einer Bestellliste, Liefer- und Rechnungsadresse einer Bestellung, Bestellpositionen oder Teilbereiche eines Berichts sein. Subressourcen sind oft ein sinnvolles Hilfsmittel, wenn Sie Operationen auf Teilen einer größeren Ressource durchführen möchten. Wir empfehlen, dass Sie dem Grundsatz folgen, dass eine Ressource eine eigene Identität haben sollte. Auch hier sehen Sie, dass es eine Entwurfsentscheidung ist, ob Sie ein fachliches Modellelement auf eine eigene Subressource abbilden oder einfach in die Repräsentation einer bestehenden Ressource einbetten.

4.2.3 Listen

Für die meisten (aber nicht für alle) Primärressourcen gibt es in der Regel auch eine zugehörige Listenressource: die Menge aller Kunden, alle Mahnungen, alle Buchungen usw. Dass diese ebenfalls eigene Ressourcen sind, bedeutet, dass auch sie eindeutig identifiziert werden und möglicherweise mehr als eine Repräsentation haben.

Eine Beispiel für eine Listenressource haben wir in unserem Beispiel gesehen: die Liste der Bestellungen. Als Reaktion auf ein GET erhalten wir eine Repräsentation der Liste zurück, über ein POST können wir eine neue Primärressource als Element der Liste anlegen.

4.2.3.1 Filter

Listenressourcen müssen nicht sämtliche Primärressourcen einer Kategorie enthalten: Auch die Liste aller Kunden aus der Region Nord, alle Produkte der

Kategorie »A« und alle Mahnungen, die im Monat Mai versandt wurden, sind jeweils Kandidaten für Ressourcen. Sie entstehen, indem man auf die Listenressourcen Filterkriterien anwendet.

4.2.3.2 Paginierung

In einer Weboberfläche sind wir es gewohnt, Suchergebnisse seitenweise präsentiert zu bekommen. Das ist in einem Webservice nicht anders: Auch hier möchte man sich gegen einen absichtlichen oder versehentlichen Zugriff auf sämtliche Elemente einer Liste absichern. Über einen Paginierungsmechanismus können Sie dafür sorgen, dass nicht alle, sondern nur eine begrenzte Menge von Daten zurückgegeben werden. Das ist sinnvoll, wenn die Datenmenge sehr groß werden kann – also praktisch immer.

4.2.4 Projektionen

Häufig ist es sinnvoll, nur einen Teil der verfügbaren Informationen für eine Primärressource oder für jedes Element einer Liste abzufragen. In diesem Fall können Sie die Informationsmenge auf eine sinnvolle Untermenge der Attribute Ihrer Objekte einschränken. Eine Hauptmotivation für diesen Ansatz ist die Reduktion der übertragenen Datenmenge.

4.2.5 Aggregationen

Im Gegensatz zu einer Projektion fassen Sie bei einer Aggregation Attribute unterschiedlicher Primär- oder Listenressourcen zusammen, um die Anzahl der notwendigen Client/Server-Interaktionen zu begrenzen.

4.2.6 Aktivitäten

Bei der Analyse finden Sie häufig noch eine Reihe von Ressourcen, die sich aus den Prozessen ergeben. So könnte ein einzelner Schritt innerhalb einer Verarbeitung oder ein ganzer Arbeitsauftrag eine eigene Ressource sein.

Häufig finden Sie diese Aktivitätsressourcen, wenn Sie die für die Umsetzung fachlicher Prozesse notwendigen Abläufe analysieren und unterstützen wollen. In der Regel starten Sie, indem Sie eine Ressource referenzieren, die es ermöglicht, eine neue Aktivität zu initialisieren. Am einfachsten können Sie sich

das anhand eines HTML-Formulars vorstellen, mit dem Sie z. B. einen neuen Kundenanlageprozess starten. Das Formular ist dabei die eine Ressource, die von allen Clients benutzt wird, die diese Aktivität initiieren wollen. Mit einem POST legen Sie eine neue Aktivitätsressource an, die dann die Daten für eine spezifische Prozessinstanz enthält (z. B. die Kundenanlage des Neukunden Claudia Musterfrau).

Während die anderen Ressourcenarten also eher den Charakter von statischen Dingen haben, bilden Aktivitätsressourcen Abläufe ab. Beides ist ein gültiger Ansatz, und in den allermeisten Fällen enthalten REST-Anwendungen Ressourcen aller Varianten.

4.2.7 Konzept- und Informationsressourcen

Aus dem *Semantic Web* stammt die Idee von Ressourcen, die nicht selbst dereferenziert werden können, sondern in denen sich nur ein reines Konzept manifestiert. Ein Beispiel wäre eine Person XY: Von dieser kann es ein Bild, eine Visitenkarte, einen Lebenslauf usw. geben. Nichts davon »ist« die Person; dennoch haben alle diese Informationen einen Bezug zur selben Identität.

Bislang haben wir unterschiedliche Formate als Repräsentationen ein und derselben Ressource verwendet. Alternativ können Sie auch eine reine Konzeptressource definieren, die auf unterschiedliche *Informationsressourcen* verweist (hinter denen sich dann das Bild, die Visitenkarte usw. verbergen).[4] Einer solchen Konzeptressource könnten Sie per POST weitere Informationsressourcen hinzufügen.

4.2.8 Evolutionäre Weiterentwicklung und YAGNI

Aus der agilen Softwareentwicklung stammt die Abkürzung YAGNI *(You ain't gonna need it)*. Damit wird eine Situation beschrieben, die den meisten Entwicklern und Architekten nur zu bekannt ist: Aufgrund nicht völlig klarer Anforderungen bauen Sie Flexibilität in ein System ein, damit Sie später sowohl mit der einen als auch mit der anderen Ausprägung einer Anforderung umgehen können, nur um letztlich festzustellen, dass Sie die Flexibilität an einer anderen Stelle gebraucht hätten. Das YAGNI-Prinzip besagt daher, dass Sie genau die Probleme lösen sollten, die Sie aktuell tatsächlich haben, und nicht solche, von denen Sie glauben, dass Sie sie später bekommen werden.

Diese Praxis sollten Sie auch bei Ihrem Ressourcenentwurf beachten. Sie können sich zunächst darauf konzentrieren, die Ressourcen zu veröffentlichen, die Sie für die Kommunikation mit der Außenwelt tatsächlich benötigen. Stellen Sie später fest, dass Sie bestimmte Anforderungen damit nicht erfüllen können, können Sie zusätzliche Ressourcen hinzufügen, ohne bestehende ändern zu müssen.[5]

4.3 Ressourcenidentifikation und URIs

Ressourcen sind identifizierbar: Über eine ID, im Web also eine URI, wird genau eine Ressource benannt. Der Umkehrschluss gilt jedoch nicht: Eine Ressource kann unter mehreren URIs gefunden werden.

Ressourcen sind den Objekten in der objektorientierten Programmierung durchaus ähnlich. Es gibt jedoch wesentliche Unterschiede in der Langlebigkeit und im Geltungsbereich: Objekte werden in der Regel im Hauptspeicher eines Programms transient, d. h. flüchtig, abgelegt – sie verschwinden, wenn das Programm beendet wird. Ressourcen im Web dagegen leben deutlich länger, idealerweise über Jahre. Referenzen in einer objektorientierten Anwendung sind nur in genau diesem Rahmen gültig – in einem System, das sich für eine weltweite Verteilung eignen soll, muss der Namensraum so organisiert sein, dass er ohne eine zentrale Stelle auskommt. Zur Umsetzung dieser Ziele haben die Designer des WWW sich auf die berühmten Schultern von Riesen gestellt und bereits bestehende Konzepte wie das Domain Name System (DNS) genutzt. Ergebnis ist die URI[6], der Mechanismus für die Identifikation von Ressourcen im World Wide Web.

Reduziert man die Ideen hinter REST auf einen einzigen Satz, ergibt sich: »Eine eigene URI für jedes Informationselement« [Megginson2007] In den folgenden Abschnitten werden wir uns URIs näher ansehen; wenn Sie die technischen Details nicht interessieren oder Sie sie schon kennen, können Sie direkt bei Abschnitt 4.4 weiterlesen.

4.3.1 URI, IRI, URL, URN, XRI?

Beginnen wir zunächst mit der Terminologie: Heißt es nun »URI« oder »URL«? Was ist eine URN? Sollten wir statt URI mittlerweile nur noch von IRI sprechen, oder sind die Tage aller dieser Varianten gezählt, weil die Zukunft der XRI gehört?

Zunächst einmal zu dem Begriff, den wir bislang nahezu ausschließlich verwendet haben: URI. Das Akronym steht für *Universal Resource Identifier* und wird im RFC 3986 [RFC3986] definiert. Die Spezifikation beschreibt dabei nicht nur HTTP-URIs: Auch per FTP erreichbare Dateien, E-Mail-Adressen und viele andere Ressourcen können über eine URI identifiziert werden. Die folgenden Beispiele (schamlos aus der Spezifikation kopiert) zeigen URIs für unterschiedliche Ressourcenschemata:

- `ftp://ftp.is.co.za/rfc/rfc1808.txt`
- `http://www.ietf.org/rfc/rfc2396.txt`
- `ldap://[2001:db8::7]/c=GB?objectClass?one`
- `mailto:John.Doe@example.com`
- `news:comp.infosystems.www.servers.unix`
- `tel:+1-816-555-1212`
- `telnet://192.0.2.16:80/`
- `urn:oasis:names:specification:docbook:dtd:xml:4.1.2`

Mit Ausnahme der letzten URI haben all diese Beispiele gemeinsam, dass sie nicht nur eine eindeutige ID festlegen, sondern auch gleichzeitig die »Adresse« (im Sinne des Protokolls, Server- oder Domainnamens). Sie sind *dereferenzierbar*. Ein System kann die Namen damit ohne weitere Suche in einem Verzeichnis verwenden, um die Ressource zu erreichen; man nennt sie daher auch URL *(Uniform Resource Locator)*.

Anders sieht es bei der letzten Variante, der URN *(Uniform Resource Name)*, aus. Eine solche URI ist nicht derefenzierbar und damit – in der Theorie – langlebiger und unabhängiger von Änderungen.

Die Unterscheidung URI, URL und URN ist allerdings eigentlich nur noch von historischer Bedeutung. Dafür gibt es mehrere Gründe. Viele Schemata lassen sich nicht eindeutig als URL oder URN spezifizieren, sondern passen für beide Varianten. Die Begriffe URN und URL werden nicht mehr empfohlen, und alle neuen Spezifikationen verwenden den Oberbegriff »URI«.[7] Schließlich haben sich HTTP-URIs in der Praxis auch dort durchgesetzt, wo man eigentlich eher URNs erwartet hätte, z. B. in Namen von XML-Namespaces. Langer Rede kurzer Sinn: Vergessen Sie URN und URL, URI ist allgemeiner und (fast) immer richtig.

Die Zeichen, die in URIs verwendet werden können, sind bewusst auf eine Untermenge des US-ASCII-Zeichensatzes eingeschränkt. Diese Restriktion gilt für IRIs nicht: *Internationalized Resource Identifiers* sind grundsätzlich identisch zu URIs, erlauben jedoch die Verwendung aller Zeichen aus dem Unicode-Zeichensatz [RFC3987]. Damit können URIs auch Sonderzeichen westlicher Sprachen (z. B. Umlaute) enthalten, sich aus dem kyrillischen Alphabet bedienen oder Sprachen unterstützen, in denen von rechts nach links geschrieben wird. Die IRI-Spezifikation enhält darüber hinaus eine Abbildung von IRIs auf URIs. Clients, die IRIs verstehen – z. B. Webbrowser, bei denen IRIs im Eingabefeld für Adressen erlaubt sind –, können diese anhand der Abbildung eindeutig in URIs umwandeln und diese dann gegebenenfalls dereferenzieren.

XRIs sind eine spezielle Form von URIs, die im Rahmen einer OASIS-Spezifikation standardisiert werden. Ziel ist dabei die Trennung von Identifikation und Netzwerklokation. Ob es sich dabei wirklich um ein Problem handelt, ist allerdings umstritten. So hat der entschiedene Einspruch der TAG (Technical Architecture Group) des W3C, der Web-Erfinder Tim Berners-Lee vorsteht, indirekt zu einer (vorläufigen) Ablehnung des XRI-Standards bei OASIS geführt. Wir betrachen XRIs daher im Folgenden nicht näher.

Für das Design von REST-konformen Anwendungen stehen ganz klar HTTP-URIs im Mittelpunkt: Alles, was identifizierungswürdig ist, sollte eine eigene URI bekommen.

4.3.2 Anatomie einer HTTP-URI

Eine HTTP-URI besteht aus einzelnen Komponenten, die durch eines der von der URI-Spezifikation definierten Trennzeichen voneinander separiert werden. Zu den wichtigsten Trennzeichen gehören »:«, »/«, »?« und »#«.

4.3.2.1 Schema und Host

Jede URI beginnt mit einem Schema, gefolgt von einem Doppelpunkt. Für jedes Schema ist definiert, wie Namen darin vergeben werden. Wir betrachten für unsere Zwecke nur die URI-Schemata »http« und »https«; innerhalb dieser Schemata erfolgt die Vergabe von Namen nach den Regeln für HTTP-URIs.

Darin folgt auf das Schema als *Authority* ein Hostname, d. h. der Name eines Rechners, einer Domäne oder einer IP-Adresse, optional gefolgt von einem Port (standardmäßig Port 80 für HTTP und Port 443 für SSL). Optional kann vor dem

Namen des Hosts noch ein Benutzername und ein Passwort angegeben werden. Wird eine nach diesem Muster aufgebaute URI dereferenziert, wird eine HTTP-Verbindung zu dem genannten Host und Port aufgebaut (oder eine bereits bestehende genutzt) und der entsprechende Request dorthin versandt.

Einige Beispiel-HTTP-URIs:

- `http://www.example.com/a/b/c`
- `http://user1:secret@www.example.com/a/b/c`
- `https://www.example.com:8443/a/b/c`
- `http://127.0.0.1`[8]`:8080/x/y/z`

Eine URI, die das HTTP-Schema verwendet, muss nicht unbedingt dereferenzierbar sein. Sie sollten jedoch die Regeln der Delegation von Verantwortung respektieren und nur URIs »unterhalb« eines Domain- oder Hostnamens anlegen, wenn Sie dazu auch die entsprechende Autorisierung haben: Auf dieser Idee der Föderation basiert die Skalierbarkeit des HTTP-Namensraumes – innerhalb jeder einzelnen Domäne ist jedes Unternehmen, jede Organisation selbst verantwortlich; für die Vergabe von Domänen selbst wiederum gibt es ebenfalls eine föderierte Vergabestruktur (InterNIC, DENIC usw.). Praktische Erfahrung zeigt jedoch, dass HTTP-URIs, die nicht dereferenzierbar sind, eine schlechte Idee sind – Sie sollten sie vermeiden.

Hinter jedem Internet-Domainnamen verbirgt sich (mindestens) eine IP-Adresse. Auf einem Rechner mit einer IP-Adresse können jedoch unterschiedliche Domainnamen abgebildet werden: Dann haben wir es mit virtuellen, nicht wirklichen Hosts zu tun. Damit solche *Virtual Hosts* funktionieren können, ist die Identifikation des Hosts, der angesprochen werden soll, in HTTP 1.1 verbindlich vorgeschrieben. Dazu wird der Hostname im Host-HTTP-Header übermittelt.

Sieht man von Schema, Hostname und Port ab, verbleiben drei Elemente einer URI: *Pfad*, *Query* und *Fragment*. In den nächsten Abschnitten werden wir uns diese näher ansehen.

4.3.2.2 Relative und absolute Pfade

Den ersten URI-Bestandteil, der auf den Hostnamen und den optionalen Port folgt, bezeichnet man als »Pfad«. Ähnlich einem Pfad zu einer Datei im

Dateisystem werden einzelne Elemente durch Schrägstriche voneinander getrennt.

Für den Client ist nicht erkennbar, welcher Bestandteil des Pfades vom Server verwendet wird, um das zuständige Programm oder die zuständige Programmkomponente zu ermitteln. Sie können also nicht erkennen, ob Anfragen an die URI

```
http://example.com/main/subsegment1/components/3
```

von einer einzigen Anwendung (z. B. einem CGI-Skript) verarbeitet werden, die beim Server als zuständig für »/main« registriert wird, oder ob sich dahinter einzelne Servlets einer Java-Webanwendung verbergen (vielleicht ein Servlet je Subsegment).

Neben absoluten Pfaden, die mit einem »//« beginnen, gibt es auch relative Pfade – ebenfalls analog zu einem Dateisystem. Sie beginnen mit einem einfachen »/« und können »..« und ».« enthalten. Damit lassen sich Ressourcen relativ zueinander adressieren. Haben Sie z. B. die Repräsentation einer Ressource im HTML-Format per HTTP GET von der URI http://example.com/A/BB/1234 geholt, so bezieht sich eine darin enthaltene Verknüpfung zu ../CC/2813 auf die URI http://example.com/A/CC/2813. Die Regeln für kompliziertere Fälle wie ../../A/./BB/./ usw. entsprechen wahrscheinlich Ihren Erwartungen; die URI-Spezifikation definiert sie inkl. aller Sonderfälle recht genau.

Relative URIs haben eine Reihe von Vorteilen. Sie können damit Platz sparen, weil sie eine feststehende, für eine Vielzahl von URIs gleiche Pfadkomponente nicht unzählige Male wiederholen müssen. Ein ganzer Baum von Ressourcen, die sich gegenseitig relativ adressieren, kann als Ganzes verschoben werden. Schließlich kann sich auch das Schema ändern, z. B. von HTTP auf HTTPS, ohne dass die Verknüpfungen davon betroffen wären.

Relative URIs sind daher generell durchaus sinnvoll. Allerdings sollte sich kein Nutzer einer Repräsentation darauf verlassen, an einer bestimmten Stelle ausschließlich absolute oder relative URIs vorzufinden: Zu einer losen Kopplung gehört die Unterstützung beider Varianten.

Die Information darüber, auf welche andere URI sich eine relative URI bezieht, kann sich entweder aus dem Kontext ergeben oder aber durch eine explizit als Basis-URI gekennzeichnete URI explizit ausgewiesen werden. Beispiele für Letzeres sind das »base«-Tag in HTML und das »xml:base«-Attribut für XML.

Darüber kann festgelegt werden, auf welche Basis sich alle relativen URIs in einer Repräsentation beziehen.

4.3.2.3 Query-Parameter

Der Pfadanteil einer URI wird durch ein (optionales) »?« von einem (ebenfalls optionalen) weiteren Anteil getrennt, der als Query bezeichnet wird. Ein Beispiel:

```
http://example.com/customers?name=Schultz&city=Rosenheim
```

Der Query-Anteil besteht aus der Zeichenkette name=Schultz&city=Rosenheim, der in diesem Fall zwei Query-Parameter enthält, die durch das Kaufmanns-Und (»&«) verkettet werden. Der Ordnung halber: Weder die REST-Dissertation noch der HTTP- oder URI-Standard definieren Query-Parameter; sie sind jedoch der Standard bei HTML-Formularen, deren Inhalte per GET an den Server übermittelt werden. Wir haben sie trotzdem an dieser Stelle aufgenommen, weil sie zumindest als Konvention häufig auch dann zum Einsatz kommen, wenn HTML keine oder nur eine untergeordnete Rolle spielt.

4.3.2.4 Matrixparameter

Matrixparameter werden relativ selten verwandt, und zwar dann, wenn nicht Schlüssel/Wert-Paare, sondern Indikatoren einer Eigenschaft oder Variante – mit anderen Worten: Boole'sche Werte – ausgedrückt werden sollen. Ein Beispiel:

```
http://example.com/customers?sortby=lastname;descending
```

Hier wird die Sortierreihenfolge durch den Matrixparameter *descending* (»absteigend«) vorgegeben. Aber auch Matrixparameter können Werte ausdrücken. Ein Beispiel aus den Yahoo!-APIs [Yahoo] ist der Zugriff auf eine Untermenge einer Liste:

```
http://social.yahooapis.com/v1/user/6677/connections;start=0;count=20
```

Ob Sie für bestimmte Informationen innerhalb der URI-Pfadanteile, Query- oder Matrixparameter verwenden, ist eine Entwurfsentscheidung innerhalb Ihrer Anwendung – aus REST- und HTTP-Sicht bleibt sie ein Implementierungsdetail.

4.3.2.5 Fragment-ID

Fragment-IDs sind ebenfalls ein optionaler Bestandteil der URI, der auf den Query-Anteil folgt und davon durch eine Raute getrennt wird:

```
http://example.com/documents/13#section12
```

Das Besondere an Fragment-IDs ist, dass das Auflösen – das Dereferenzieren – immer clientseitig erfolgt. Bekanntestes Beispiel für die Verwendung von Fragmenten ist das Zusammenspiel mit dem HTML-Anchor-Element (»<a …/>«). Sehen wir uns als Beispiel ein einfaches HTML-Dokument an:

```
<html>
  <body>
    <p><a name=»intro» />Dies ist die Einleitung</p>
    <p><a name=»detail» />Hier kommt die detaillierte Beschreibung …
</p>
  </body>
</html>
```

Wird die URI – z. B. *http://example.com/document#detail* – von Ihrem Webbrowser dereferenziert, so wird das vollständige Dokument übertragen und erst danach so weit gescrollt, bis das a-Element mit dem Namen »detail« sichtbar ist. Diese clientseitige Auflösung gilt allgemein für alle Arten von URIs; die spezifische Abbildung ist vom Medientyp (im Fall unseres Beispiels HTML) abhängig. Eine Fragment-ID wird also niemals zum Server übertragen, und es ist Sache des Clients, sie korrekt aufzulösen.[9]

4.3.3 URI-Templates

URI-Templates sind recht verbreitet und mittlerweile durch einen offiziellen Standard definiert [RFC6570]. Mit ihrer Hilfe kann man Muster für URIs beschreiben, bestehend aus fixen und variablen Anteilen. Die in der Spezifikation vorgesehenen Mittel sind sehr umfassend, in der Praxis eingesetzt wird jedoch nur eine recht überschaubare Untermenge. So finden Sie häufig eine Darstellung von URIMengen in folgender Form:

```
http://example.com/orders/{id}
```

Damit ist die Menge sämtlicher URIs gemeint, die aus dem Präfix *http://example.com/orders/* und einer angehängten ID bestehen.

Eine häufige Verwendung von URI-Templates ist die Dokumentation sehr vieler ähnlicher Ressourcen. Viele »REST-APIs« sind eigentlich URI-APIs: Sie definieren eine Menge von Ressourcenmustern und die Bedeutung der Standard-HTTP-Methoden, wenn man sie auf solche Ressourcen anwendet. Da solche APIs häufig (und berechtigt) als wenig RESTful kritisiert werden, betrachten manche REST-Experten URI-Templates als Anti-Pattern. Wir teilen diese Ansicht nicht, da sie sich sehr gut auch in Antworten auf HTTP-Anfragen

einbetten lassen und damit bei JSON, XML oder anderen Formaten die Rolle der Formulare bei HTML einnehmen können.

4.4 URI-Design

Ein letztes Mal möchten wir betonen, dass das Design von URIs nicht so wichtig ist, wie es am Anfang erscheint. Aus REST-Sicht ist der Einsatz von Hypermedia der Weg, auf dem Systeme über die »richtigen« URIs informiert werden, nicht nur mithilfe der im letzten Abschnitt erwähnten URI-Templates – mehr dazu erfahren Sie in Kapitel 6. Sie sollten auf jeden Fall vermeiden, dass Clients mit String-Operationen URIs zusammensetzen, denn dazu müssen diese intime Kenntnis über Server-Interna haben.

4.4.1 URI-Entwurfsgrundsätze

Trotzdem sind elegante URI-Schemata oft Zeichen einer sinnvollen Ressourcenstruktur. Wenn URIs sinnvoll gestaltet sind, erleichtert dies einem Entwickler, der ein REST-API nutzen möchte, die Arbeit; Endanwender schätzen URIs, die man sich merken kann; auch die Möglichkeit zum »Hacking« von URIs – das manuelle Bearbeiten der URI, z. B. durch Abschneiden der Elemente hinter den einzelnen Schrägstrichen – ist durchaus ein Vorteil. Jakob Nielsen, der sich mit Usability-Aspekten beschäftigt, betrachtet URIs unter anderem deswegen als Elemente des Benutzerschnittstellenentwurfs [Nielsen1999a] Im Folgenden möchten wir aus diesem Grund zuerst den Aufbau von URIs beschreiben und danach Hinweise geben, wie ein sinnvoller Entwurf der URI-Schemata Ihrer Anwendung aussehen kann.

URIs und REST

Die wichtigste Grundregel für das URI-Design beim Entwurf von REST-Anwendungen lautet: Bewerten Sie das Design nicht zu hoch; es sollte sinnvoll sein, aber nicht perfekt (das ist ohnehin nicht zu schaffen). URIs sind vergleichbar mit den Variablen-, Methoden- oder Klassennamen in der Programmierung: Auch hier lohnt es sich, über »gute« Namen nachzudenken. Für Compiler und Ablaufumgebung jedoch sind sie letztendlich unerheblich.

Allgemeines

URIs identifizieren Ressourcen; Ressourcen sind »Dinge«, Substantive, keine Aktionen oder Verben. Wenn Sie ein korrektes Ressourcendesign durchgeführt haben, ergeben sich passende Namen in der Regel von allein. Dennoch möchten wir kurz einige Beispiele für URIs vorstellen, die aufgrund ihrer Struktur oder ihres Namens eine Indikation für ein Problem sein können.

Recht offensichtlich ist die Verwendung von Verben in der URI:

```
http://example.com/customers/create?name=XYZ
```

Hier ist einiges durcheinandergeraten: Das Verb »create« hat in der URI auf den ersten Blick nichts zu suchen, und die Struktur suggeriert, dass hier über die URI die Aktion identifiziert wird, nicht über das HTTP-Verb.

Ein weiteres Beispiel:

```
http://example.com/RequestProcessor?method=build&p1=x&p2=z
```

In diesem Fall sehen wir ein weiteres Muster, das auf ein Problem hinweist: Die Methode wird als Wert eines Parameters übergeben. Hier ist der »Request-Processor« vermutlich ein Gateway, das eine RPC-orientierte Interaktion durch HTTP »tunnelt«.

Ihre URIs sollten Substantive enthalten; wenn Sie sich vorstellen können, dass Sie einen Link darauf einem Kollegen per E-Mail zusenden, ist das ein gutes Zeichen. Bei URIs, die – wie es gedacht ist – Ressourcen identifizieren, können Sie sich darüber hinaus leicht vorstellen, wie die unterschiedlichen HTTP-Methoden wirken: Ein PUT aktualisiert, ein DELETE löscht, ein GET holt Informationen. Ein Beispiel für eine solche URI ist die folgende:

```
http://example.com/countries/049
```

Hierarchie

Eine naheliegende und vergleichsweise offensichtliche Regel für ein transparentes und nachvollziehbares URI-Design ist die Abbildung von hierarchischen Beziehungen auf die Pfadelementstruktur der URI. Ein Beispiel dafür ist die Organisationsstruktur eines Unternehmens. Die Gruppe »Netzwerke« in der Abteilung »Support« der Hauptabteilung »Betrieb« im Vorstandsbereich »IT« könnte über eine URI der Form

```
http://example.com/Organisation/IT/Betrieb/Support/Netzwerke
```

abgebildet werden.[10] Ähnlich wie bei einer Pfadangabe in einem Unix- oder Windows-Dateisystem »sieht« der Anwender oder Programmierer einer solchen URI die Hierarchie regelrecht an. Als Konsequenz daraus, dass Webserver

statische Dateien aus einem Dateisystem, das mit Ordnern strukturiert ist, ebenfalls direkt abbilden, erwartet man darüber hinaus, dass auch übergeordnete Elemente sinnvoll sind. In unserem Beispiel könnte ein Endanwender durchaus auf die Idee kommen, den letzten Teil des Pfades (»/Netzwerke«) von Hand zu löschen, falls er etwas zum Support für ein anderes Thema erfahren möchte.

Wie in vielen anderen Situationen beim Design von REST-Anwendungen ist es wichtig, dieses Verhalten zu kennen und sich darauf vorzubereiten, es jedoch selbst (vor allem in den eigenen programmatischen Clients) zu vermeiden. Mit anderen Worten: Wenn Sie der Entwickler des Servers und damit der Designer der URI-Struktur sind, sollten Sie dafür sorgen, dass auch ein GET auf http://example.com/Organisation/IT/Betrieb/Support ein sinnvolles Ergebnis zurückliefert. In Ihrem Client sollten Sie eine solche Annahme jedoch auf keinen Fall treffen. Diese Toleranz gegenüber dem, was Sie als Eingabe von anderen erhalten, bei gleichermaßen striktem Einhalten der Regeln, wenn Sie selber Daten an andere Systeme liefern, trägt fundamental zur Stabilität eines verteilten Systems bei und ist auch bekannt als das Robustheitsprinzip oder auch als das Postel'sche Gesetz[11].

Externe und interne Schlüssel

Wenn Sie eine Listenressource identifizieren möchten, können Sie selbst bestimmen, wie Sie den Namen wählen – die Menge der Konzepte, die Sie auf Listen abbilden, wird in der Regel sowohl begrenzt als auch gut verstanden sein, sodass es nicht schwerfällt, einen Namen zu finden. Abgebildet auf URIs definieren Sie damit vielleicht Ressourcen mit den Namen /customers, /orders oder /products.

Für die Identifikation der einzelnen Primärressourcen, also für jeden einzelnen Kunden, jede Bestellung oder jedes Produkt, müssen Sie jedoch statt eines Namens ein Muster definieren. Dabei stellt sich die Frage, ob Sie einen fachlichen Schlüssel wie zum Beispiel einen Namen, eine Postleitzahl oder eine Steuernummer oder einen technischen Schlüssel, der keine offensichtliche Bedeutung hat, wählen sollten.

Die Frage, ob man als Primärschlüssel fachliche oder technische Schlüssel wählen sollte, ist bereits aus der Datenbankmodellierung bekannt [Ambler2005]. Der größte Vorteil fachlicher Schlüssel ist, dass der Abfragende – sei es ein Endanwender oder der Entwickler einer Anwendung – den Schlüssel

wahrscheinlich schon kennt und kein neuer eingeführt werden muss. Allerdings gibt es eine ganze Reihe von Nachteilen: Ändern sich Schlüsselwerte, betrifft dies auf einmal auch die Identität. Technische Schlüssel dagegen können konstant bleiben; die fachlichen Schlüsselattribute können zusätzlich mitgeführt werden. Es gibt allerdings auch Fälle, in denen ein technischer Schlüssel keinen Mehrwert bringt: Es lohnt sich zum Beispiel kaum, einen Schlüssel wie eine Postleitzahl noch einmal hinter einem technischen Schlüssel zu verbergen.

Für das Design von URIs lassen sich diese Erkenntnisse eingeschränkt übertragen. Es ist eine gute Idee, in den URIs nur Schlüssel zu verwenden, die dauerhaft konstant bleiben (siehe auch Abschnitt 4.4.3). Andererseits unterstützen URIs ein Redirect, also eine Umleitung, falls ein Ressourcenumzug notwendig wird. Darüber hinaus kann ein und dieselbe Ressource durchaus über mehrere URIs identifiziert werden, sodass Sie auch mehrere Identifikationswege ermöglichen können. Hierbei sollten Sie allerdings die Auswirkungen auf das Caching berücksichtigen (siehe Kapitel 10).

Pro und Contra Query-Parameter

Für den Entwurf von URIs für Filter haben Sie verschiedene Möglichkeiten.

Am naheliegendsten ist die Verwendung von Query-Parametern. Für eine Liste von Kunden unter /customers, die man nach allen Kunden aus Deutschland filtern möchte, könnte eine Beispiel-URI folgendermaßen aussehen:

```
http://example.com/customers/?country=Germany
```

Mehrere Query-Bestandteile werden mit einem Kaufmanns-Und miteinander verkettet:

```
http://example.com/customers/?country=Germany&type=A&year=2009
```

Auch hier erfüllen Sie mit einem solchen URI-Design die Erwartungen, die Nutzer Ihrer Ressourcen an URIs haben: Lässt man den Query-String (die Summe der Query-Parameter) weg, gelangt man zur ungefilterten Liste. Darüber hinaus spricht für diesen Entwurf, dass ein HTML-Formular verwendet werden kann, um eine solche URI zu konstruieren.

Das stärkste Argument gegen Query-Parameter ist mittlerweile historisch: Einige Cache-Implementierungen, allen voran der weitverbreitete Squid [Squid], verweigerten in der Standardkonfiguration das Caching von Inhalten, deren URI ein »?« enthält. Mit der Version 2.7, die seit Mai 2008 verfügbar ist, wurde dies

geändert, sodass Sie nur noch in seltenen Fällen damit rechnen müssen, dass das Problem noch besteht.

4.4.2 REST aus Versehen

Mit dem Begriff »Accidentally RESTful« – frei übersetzt: REST-konform aus Versehen – bezeichnet man die Situation, in der man dem Schnittstellenentwurf relativ klar ansehen kann, dass der Designer REST nicht verstanden hat, aber sich zufällig dennoch an die Vorgaben hält [Baker2005]. Das ist in der Regel immer dann der Fall, wenn in der URI ein Methodenname enthalten ist, die aufgerufene Methode aber »safe« im HTTP-Sinne ist.

Am besten lässt sich dies an einem Beispiel zeigen. Vergleichen Sie die beiden folgenden URIs:

- `http://example.com/customerservice?operation=findCustomer&id=1234`
- `http://example.com/customers/1234`

Ganz offensichtlich erkennt man in der ersten URI, dass der Designer »in Operationen denkt«: findCustomer ist die Operation, die aufgerufen wird, 1234 der erste Parameter. Die zweite URI dagegen identifiziert klar eine Ressource – den Kunden. Man könnte daraus schließen, dass die erste URI nicht REST-konform ist, die zweite dagegen schon.

Tatsächlich jedoch ist eine solche Aussage Unsinn: Die URI ist aus Sicht von REST nur eine ID, völlig unabhängig davon, aus welchen Buchstaben sie zusammengesetzt ist. Wenn Sie sich die URIs vorstellen, die sich für die Abfrage der Daten einiger anderer Kunden ergeben würden, dann stellen Sie fest, dass auch in diesem Entwurf jeder Kunde eine eigene URI bekommt. Sie sieht zwar nicht aus wie die ID einer Ressource, aber das kann uns im Prinzip egal sein.

Es ist aber sehr wahrscheinlich, dass dem Designer des ersten URI-Schemas die REST-Ideen nicht geläufig waren und sich auch andere, ähnliche Muster dort finden. Auch hierzu ein Beispiel:

```
http://example.com/customerservice?operation=deleteCustomer&id=1234
```

Offensichtlich können in diesem Entwurf alle Arten von Operationen über ein GET auf eine entsprechend konstruierte URI aufgerufen werden – auch solche, für die gemäß Spezifikation ein POST, PUT oder DELETE angemessen wäre. Und

unterschiedliche Operationen, die auf dieselbe Ressource wirken, werden auf unterschiedliche URIs abgebildet anstatt auf dieselbe.

Das Auftreten von Operationsnamen in URIs ist also ein Warnzeichen, ein »Design Smell«. In den meisten Fällen hat der Designer der Schnittstelle die REST-Prinzipien nicht verstanden oder – in seltenen Fällen – bewusst ignoriert. Einige Operationen können dabei so abgebildet werden, dass die REST-Prinzipien nicht verletzt werden: REST-Konformität aus Versehen.

4.4.3 Stabile URIs

Von Web-Erfinder Tim Berners-Lee stammt der Satz »Cool URIs don‘t change« – coole URIs ändern sich nicht [BernersLee1998]. URIs, die Sie an die Außenwelt – d. h. Ihre Clients – weitergeben, sind Teil Ihrer APIs. Und genauso wenig wie Sie bei der Entwicklung einer Bibliothek, die von Ihnen unbekannten Benutzern verwendet wird, einfach den Namen einer Operation ändern würden, genauso wenig sollten Sie eine URI, die einmal gültig war, ins Leere laufen lassen.

Das HTTP-Protokoll unterstützt dies durch zwei wesentliche Aspekte. Zum einen können Sie mit einer Umleitung, einem *Redirect*, eine einmal veröffentlichte URI bei der Anfrage auf eine andere umleiten. Dies kann permanent oder optional geschehen, außerdem können Sie den Client auch über mehrere alternative URIs informieren. Zum anderen ist auch der Fall, dass eine URI nicht mehr sinnvoll aufgelöst werden kann, Bestandteil des Protokolls. Dazu können Sie dem Client über den Statuscode »410 Gone« mitteilen, dass eine URI zwar früher einmal existierte, aber nun nicht mehr verfügbar ist. Als letzten Ausweg kann der Server dem Client mitteilt, dass es die angeforderte Ressource nicht gibt (»404 Not Found«).

Eine detaillierte Auflistung der einzelnen HTTP-Statuscodes, die Sie für diesen Zweck verwenden können, finden Sie im Anhang A.

4.5 Zusammenfassung

Ressourcen bilden die zentrale Abstraktion im REST-Architekturstil. Sie sind eindeutig identifizierbar, und die Interaktion mit ihnen erfolgt immer über den Austausch von Repräsentationen. Im Web werden für die Identifikation von Ressourcen URIs verwendet; ein sinnvoller Entwurf der URI-Struktur ist nützlich, aus REST-Sicht aber weniger wichtig, als dies auf den ersten Blick erscheinen mag.

Eine bestehende fachliche Domäne auf einen Schnittstellenentwurf abzubilden, der sich an Ressourcen orientiert, erfordert für jemanden, der in der Vergangenheit klassenorientierte Schnittstellen gewohnt war, ein Umdenken. Kritisch ist dabei, Ressourcen nicht als Konzepte einer Persistenzschicht, sondern als übergreifende, nach außen sichtbare Anwendungskonzepte zu begreifen.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

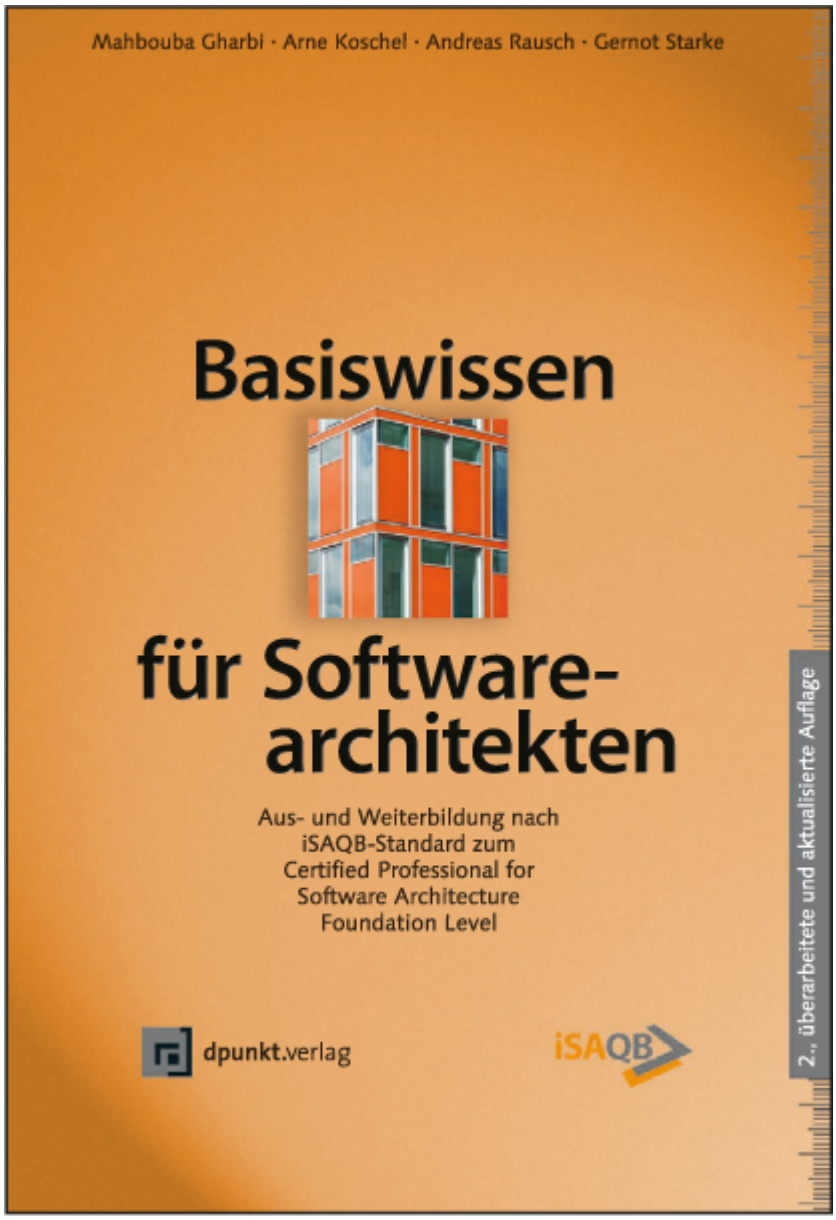
X

Z

Ziffern

Mahbouba Gharbi · Arne Koschel · Andreas Rausch · Gernot Starke

Basiswissen für Softwarearchitekten



2., überarbeitete und aktualisierte Auflage
2015, 220 Seiten, Festeinband
€ 32,90 (D)
ISBN 978-3-86490-165-2 (Buch)
ISBN 978-3-86491-621-2 (PDF)
ISBN 978-3-86491-622-9 (ePub)

Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture – Foundation Level

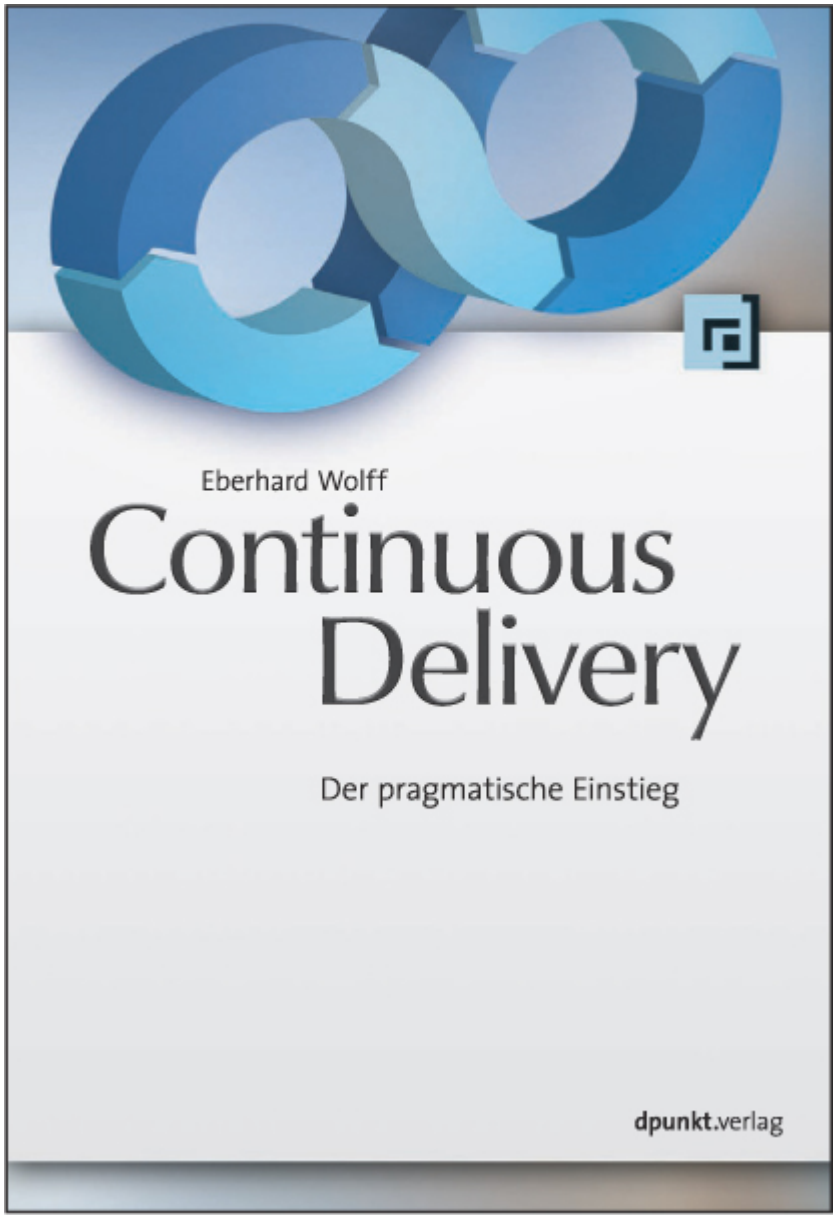
Dieses Buch behandelt die wichtigen Begriffe und Konzepte der Softwarearchitektur und beschreibt darauf aufbauend die grundlegenden Techniken und Methoden für den Entwurf, die Dokumentation und die Qualitätssicherung von Softwarearchitekturen. Ausführlich behandelt werden zudem die Rolle, die Aufgaben, das Umfeld und die Arbeitsumgebung des Softwarearchitekten, ebenso dessen Einbettung in die umfassende Organisations- und Projektstruktur.

Die überarbeitete und aktualisierte 2. Auflage ist konform zum aktuellen iSAQB-Lehrplan Version 2.93 und beinhaltet jetzt auch Beispielübungen.

Wieblinger Weg 17 · 69123 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
www.dpunkt.de

Eberhard Wolff

Continuous Delivery



1. Auflage
2015, 264 Seiten, Broschur
€ 34,90 (D)
ISBN 978-3-86490-208-6 (Buch)
ISBN 978-3-86491-589-5 (PDF)
ISBN 978-3-86491-590-1 (ePub)

Der pragmatische Einstieg

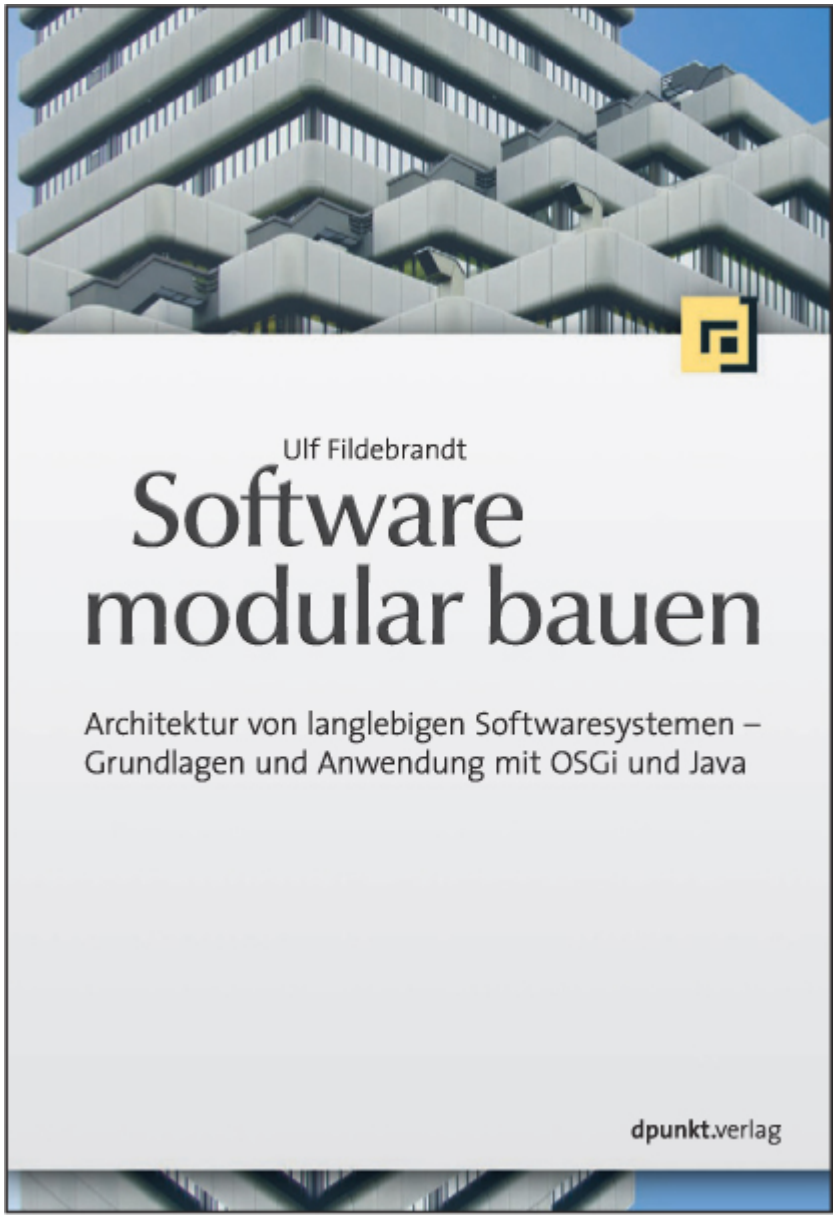
Dieses Buch erläutert, wie eine Continuous-Delivery-Pipeline praktisch aufgebaut wird und welche Technologien dazu eingesetzt werden können. Dabei geht es nicht nur um das Kompilieren und die Installation der Software, sondern vor allem um diverse Tests, die die Qualität der Software absichern.

»Wolffs Beschreibung eines möglichen Continuous-Delivery-Prozesses ist angenehm praxisnah gehalten. Von der Lektüre profitieren Software-Entwickler und Betriebs-IT-Leute ebenso wie Manager.« c‘t 7/15

Wieblinger Weg 17 · 69123 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
www.dpunkt.de

Ulf Fildebrandt

Software modular bauen



2012, 332 Seiten, Broschur
€ 39,90 (D)
ISBN 978-3-86490-019-8 (Buch)
ISBN 978-3-86491-182-8 (PDF)
ISBN 978-3-86491-183-5 (ePub)

Architektur von langlebigen Softwaresystemen – Grundlagen und Anwendung mit OSGi und Java

Dieses Buch schlägt eine Brücke zwischen den abstrakten Patterns und Konzepten der Softwareentwicklung einerseits und der realen Implementierung. Das Buch beginnt auf der Ebene des Codings, danach stehen Komponenten und die Regeln ihrer Zusammenstellung im Vordergrund. Anschließend betrachtet

der Autor die Architektur des Systems und von dessen Schichten. Er erklärt dabei die Konzepte anhand von kontinuierlich erweiterten Beispielen auf Basis von OSGi und Java. So werden die Prinzipien modularer Systeme sichtbar, mit denen man Entkopplung, Erweiterbarkeit und Einfachheit von Software erreichen kann.

Wieblinger Weg 17 · 69123 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
www.dpunkt.de