7 Repräsentationsformate

Ressourcen und Repräsentationen sind wie zwei Seiten derselben Medaille – diese Dualität haben wir bereits in Kapitel 4 ausführlich diskutiert. Wie genau sehen aber Repräsentationen von Ressourcen technisch aus? Wie steht es mit der Hypermedia-Unterstützung jenseits von HTML? Welche Alternativen haben wir hier als Entwickler und Architekten, welche Vorteile gewinnen wir durch die Entscheidung für ein bestimmtes Repräsentationsformat und welche Nachteile handeln wir uns vielleicht andererseits ein? Darum soll es in diesem Kapitel gehen.

7.1 Formate, Medientypen und Content Negotiation

Die Repräsentationen von Ressourcen, die beim Zugriff auf die URIs ausgetauscht werden, können unterschiedliche Formate haben. Der Client gibt als Teil der Anfrage in einem »Accept«-Header an, welche Typen von Inhalten (*Media Types*) er akzeptiert. Am besten sehen wir das an einem GET-Request:

```
GET /someURI HTTP/1.1
Host: example.com
Accept: image/gif, image/x-xbitmap, image/png, */*;q=0.1
Accept-Encoding: bzip2, gzip, deflate, identity
Accept-Charset: utf-8, iso-8859-1, macintosh, windows-1252, *
Accept-Language: en;q=0.99, de;q=0.94
```

Die Parameter bei den Accept-Headern für Format, Codierung, Zeichensatz und Sprache wie z. B. »q=0.94« geben an, wie die Alternativen untereinander zu gewichten sind.

Antworten sowie Anfragen, die einen Inhalt transportieren (wie PUT und POST), geben mithilfe des Content-Type-Headers an, welcher Repräsentationstyp tatsächlich gewählt wurde. Antworten auf einen GET-Request haben wir schon gesehen:

```
HTTP/1.1 200 OK
Date: Thu, 13 Oct 2005 22:38:17 GMT
Server: Apache/2.0.52 (Unix)
```

```
Content-Length: 34026
Content-Type: application/xml

<?xml version="1.0" standalone="yes" ?>
...
</xml>
```

Die Typen werden dabei nach RFC 2046 [RFC2046] codiert, für JSON z. B. kommt hierbei entweder »application/json« oder ein dem eingesetzten Format zugeordneter, idealerweise offiziell bei der IANA registrierter Medientyp zum Einsatz.[1]

Die Entwicklung von REST-Anwendungen besteht zu einem erheblichen Teil aus der Auswahl und detaillierten Festlegung der Nutzung von Repräsentationstypen. Über die Mechanismen zur Aushandlung der unterstützten MIME-Types – die Content Negotiation – können die Kommunikationspartner sich dynamisch auf eine geeignete Repräsentation einigen.

Je allgemeingültiger das Format Ihrer Repräsentationen ist, desto mehr potenzielle Clients können damit etwas anfangen. Dem steht entgegen, dass ein allgemeingültiges Format weniger spezifisch auf Ihre konkreten Anforderungen zugeschnitten ist.

Ob Sie ein eigenes Format erfinden oder auf ein Standardformat setzen, ist also eine typische Entwurfsentscheidung, bei der Sie die Vor- und Nachteile gegeneinander abwägen müssen. Erfreulicherweise bietet REST Ihnen die Möglichkeit, das eine zu tun, ohne das andere zu lassen: Sie können für Ihre Ressourcen durchaus sowohl ein standardisiertes als auch ein spezifisches Format anbieten.

Aber auch wenn Sie sich für ein standardisiertes Format entscheiden, müssen Sie Ihre spezifischen Anforderungen geeignet darauf abbilden. In den folgenden Abschnitten werden wir uns deshalb mit den populärsten Standardformaten für RESTful HTTP beschäftigen.

7.2 JSON

Wenn Sie schon die erste oder zweite Auflage dieses Buches kennen, ist es Ihnen sicher sofort aufgefallen: Anstelle von XML haben wir für unser OrderManager-Beispiel in dieser Auflage die JavaScript Object Notation (JSON) [RFC7159] verwendet. Wir haben die Frage nach dem am besten geeigneten Repräsentationsformat innerhalb des Autorenteams lange diskutiert und uns letztlich dafür entschieden, dem allgemeinen Trend zu folgen und dieses Mal JSON den Vorzug zu geben[2].

1. Die IANA veröffentlicht eine Liste der offiziell registrierten Medientypen im Web [IANAMIME]. Die Registrierung neuer Typen steht prinzipiell jedem Anwender oder Hersteller offen, der dazu erforderliche offizielle (und bewusst etwas steinige) Weg ist unter [RFC4288] dokumentiert.
2. Das bedeutet aber nicht, dass XML weniger geeignet ist. Dazu später mehr.

JSON stammt zwar ursprünglich aus dem JavaScript-Umfeld, wird aber mittlerweile in praktisch jeder Programmierumgebung unterstützt. JSON wird häufig mit XML verglichen, verfolgt allerdings andere Ziele: Der Fokus liegt ausschließlich auf Datenstrukturen, nicht auf Textdokumenten. Darüber hinaus unterstützt JSON weder Namensräume noch eine standardisierte schemabasierte Validierung[3]. Dafür ist JSON allerdings auch deutlich einfacher, wie die Beispiele in Kapitel 3 sicherlich gezeigt haben.

JSON ist natürlich besonders in JavaScript, aber auch in anderen dynamischen Programmiersprachen sehr einfach zu verarbeiten. Auch für .NET und die JVM existieren entsprechende Bibliotheken. Für JSON gibt es auch einen offiziellen Medientyp: »application/json«, standardisiert in [RFC4627]. Ähnlich wie z. B. »application/xml« sagt er aber wenig über die Bedeutung der Inhalte aus, Sie können also aus dem Medientyp nicht auf die Semantik schließen. Darüber hinaus gibt es in JSON selbst keine standardisierte Darstellung von Links, was primär daran liegt, dass für Links kein spezieller Datentyp in JavaScript existiert.

Aus diesem Grund haben sich in jüngerer Zeit eine Reihe auf JSON basierender Repräsentationsformate mit besonderem Schwerpunkt auf der Unterstützung von Hypermedia-Elementen entwickelt, deren Autoren sich zum Ziel gesetzt haben, hier Abhilfe zu schaffen. Wir werden uns in den folgenden Abschnitten einige der zurzeit »heißesten« Kandidaten etwas genauer ansehen.

7.2.1 HAL

HAL (Hypermedia Application Language) [HAL] ist das älteste der JSON-basierten Hypermedia-Formate. Autor Mike Kelly hat das Format ursprünglich 2011 als universelles Hypermedia-Format für XML[4] und JSON vorgeschlagen, letztendlich aber nur die JSON-Variante weiter ausspezifiziert, die mittlerweile eine hohe Stabilität erreicht hat.

Der wichtigste Bestandteil der Spezifikation ist ein JSON-Property namens »_links«. Der Name beginnt nicht aus Zufall mit einem Unterstrich: Dadurch sollen Kollisionen mit schon existierenden Namen vermieden werden, was die Anreicherung bestehender JSON-Dokumentformate und -Dokumente mit Links erleichtert. Innerhalb des »_links«-Properties können mit Relationen klassifizierte Verknüpfungen untergebracht werden. Darüber hinaus spezifiziert HAL mit dem »_embedded«-Property, wie die Repräsentationen verknüpfter Ressourcen eingebettet werden können, wenn so die Anzahl von Client/Server-Interaktionen minimiert werden soll.

---

3. Sie können jedoch JSON Schema [JSONSchema] nutzen, für das auch in einigen Programmiersprachen Validatoren existieren.
4. Der aktuelle Internet-Draft zu HAL [Kelly2013] von Oktober 2013 nennt das Format inzwischen »JSON Hypertext Application Language« und enthält keine Referenz auf XML mehr.

»CURIE« ist die Abkürzung für »Compact URI Expression« und Teil der RDFa-Spezifikation. Mit einer CURIE können Sie den Vorteil der Eindeutigkeit von URIs erreichen, ohne dabei jedes Mal dieselbe Zeichenkette schreiben oder lesen zu müssen. Dazu wird ein Präfix mit einer URI verknüpft und danach als Abkürzung verwendet. Wenn Sie mit Namespaces und »QNames« in XML vertraut sind: CURIEs sind ungefähr das Gleiche, allerdings syntaktisch weniger restriktiv, weil der Name, den Sie nach dem Präfix definieren, kein gültiger XML-Elementname sein muss. HAL benutzt CURIEs für Link-Relationen, um die Namen von Attributen eindeutig und trotzdem lesbar zu halten.

Das folgende Beispiel für eine Collection von Bestellungen aus der HAL-Dokumentation illustriert die wesentlichen Features des Formats:

```
{
     "currentlyProcessing": 14,
     "shippedToday": 20,
     "_links": {
        "self": { "href": "/orders" },
        "next": { "href": "/orders?page=2" },
        "curies": [{ "name": "ea",
               "href": "http://example.com/docs/rels/{rel}",
               "templated": true }],
        "ea:admin": [{
            "href": "/admins/2",
            "title": "Fred"
        }, {
            "href": "/admins/5",
            "title": "Kate"
        }],
        "ea:find": {
            "href": "/orders{?id}",
            "templated": true
        },
    },
    "_embedded": {
        "ea:order": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "ea:basket": { "href": "/baskets/98712" },
                "ea:customer": { "href": "/customers/7809" }
            },
            "total": 30.00,
            "currency": "USD",
            "status": "shipped"
        }, {
            "_links": {
                "self": { "href": "/orders/124" },
                "ea:basket": { "href": "/baskets/97213" },
                "ea:customer": { "href": "/customers/12369" }
            },
            "total": 20.00,
```

```
            "currency": "USD",
            "status": "processing"
        }]
    }
}
```

Nach den Beispielattributen (»currentlyProcessing«, »shippedToday«) finden Sie unter »self« und »next« die üblichen Links auf die Ressource selbst und ihren logischen Nachfolger. Im »curies«-Array wird nur ein Präfix (»ea«) definiert. Eine Besonderheit von CURIEs in HAL ist, dass diese einen Template-Parameter enthalten können. Dieser ist aber auf den Wert »rel« beschränkt, was für den Anwendungsfall ausreicht: So kann der Wert der Link-Relation nicht nur am Ende, sondern auch an anderer Stelle in die URI eingesetzt werden. Was recht theoretisch klingt, wird an einem Beispiel klarer: Wird in der nächsten Zeile »ea:admin« verwendet, setzt ein konformer HAL-Parser den Wert »admin« in den Parameter »rel« ein, expandiert die Schablone also zu »*http://example.com/docs/rels/admin*«. Für die unter »ea:admin« in einem Array aufgelisteten Links gilt übrigens dieselbe Relation. So werden wiederum Redundanzen vermieden.

In den eigentlichen Links lassen sich nicht nur feste Links, sondern – konform zur Spezifikation für URI-Templates – auch Schablonen für die Konstruktion von URIs unterbringen. Die Relation »ea:find« demonstriert das anhand einer Schablone, die (vermutlich) eine Suche nach der ID ermöglicht. Dazu müssten wir die Dokumentation der Link-Relation zurate ziehen. Guter Stil ist es, wenn sich diese hinter der URI »http://example.com/docs/rels/find« befindet.

Unter »_embedded« schließlich finden wir eingebettete Ressourcen, in diesem Fall eine Liste mit zwei Bestellungen, deren interne Struktur wiederum beliebiges JSON, garniert mit einem »_links«- und potenziell weiteren geschachtelten »_embedded«-Elementen sein kann. Die aktuelle Version der Spezifikation enthält hierzu sogar eine Beschreibung, wie ein Server oder Intermediary eine verknüpfte Ressource automatisiert einbetten kann (das sog. »Hypertext Cache Pattern«).

Der größte Nachteil von HAL besteht darin, dass es bei HAL (zumindest in der derzeitigen Version) keine Möglichkeit gibt, Links mit zusätzlichen Informationen über mögliche Aktionen (wie beispielsweise den darauf erlaubten HTTP-Operationen oder den bei POST- bzw. PUT-Requests zu sendenden Daten) zu annotieren. Diese bewusste Entscheidung des Autors hält das Format einerseits einfach, erfordert aber andererseits mehr Wissen auf der Seite des Clients, der solche Informationen der Dokumentation zu den Link-Relationen entnehmen muss.[5]

5. Wie Sie in Kapitel 14 sehen werden, haben wir uns für dieses Buch entschieden, HAL wegen seiner Schlichtheit dennoch zu verwenden und einfach selbst um ein entsprechendes Attribut zu erweitern. Das können Sie in Ihren eigenen Anwendungen natürlich auch jederzeit tun – die meisten JSON-Parser ignorieren unbekannte Attribute einfach, sodass auch ein erweitertes Dokument weiterhin HAL-konform ist und von generischen HAL-Clients verstanden wird.

Für HAL gibt es diverse Bibliotheken, die es erlauben, vergleichsweise einfach auf die spezifikationskonform gestalteten Links zuzugreifen. Darüber hinaus erlaubt Kellys Open-Source-HAL-Browser [HALBrowser], ein beliebiges HAL-API interaktiv zu untersuchen.

Der MIME-Type für HAL lautet »application/hal+json«.

7.2.2 Collection+JSON

Mike Amundsen, wie Mike Kelly ebenfalls eine bekannte Größe in der REST-Community, hat mit Collection+JSON [CollectionJSON] einen etwas anderen Weg beschritten. Zwar gibt es auch dort keine explizite Möglichkeit, Aktionen auszudrücken, aber dafür kann sich ein Client bei einem JSON-Dokument, das diesem Format entspricht, auf eine deutlich rigidere Struktur verlassen. Dokumente enthalten – wie der Name des Formats schon nahelegt – eine Liste von Einträgen, wobei darin natürlich auch nur ein einzelnes Element enthalten sein kann. Die Interaktion eines Clients mit Ressourcen, die Collection+JSON als Repräsentation liefern, folgt dann einer klar definierten Semantik. So können Elemente hinzugefügt, entfernt, aktualisiert und gelesen sowie Suchabfragen ausgeführt werden. Welche HTTP-Verben dazu jeweils zu verwenden sind, ergibt sich implizit aus dem Kontext bzw. ist in der Spezifikation beschrieben.

Um den Client darüber zu informieren, welche Daten zum Erzeugen oder Schreiben bzw. Ändern eines Elements übermittelt werden müssen, kann die Repräsentation einer Collection optional ein sogenanntes Template enthalten. Abhängig vom aktuellen Anwendungszustand kann dieses Template durchaus bei verschiedenen Interaktionen strukturell und inhaltlich immer wieder unterschiedlich aussehen oder etwa bestimmte Attribute mit Werten vorbelegen. Dadurch ist dieser Mechanismus recht flexibel.

Etwas Ähnliches gilt für Queries: Auch dafür kann die Repräsentation der Collection eine Vorlage in Form eines URI-Templates enthalten. Der Client muss dann nur noch die Query-Parameter aus dem Template mit Werten belegen und einen GET-Request auf die URI ausführen.

Das folgende Beispiel zeigt den Aufbau eines Collection+JSON-Dokuments:

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/friends/",

    "links" : [
      {"rel" : "feed", "href" : "http://example.org/friends/rss"}
    ],

    "items" : [
      {
        "href" : "http://example.org/friends/jdoe",
```