REST-Unterstützung in Rails 2.x

RESTful Web Services mit Rails

Stefan Tilkov

In den letzten zwei bis drei Jahren setzt sich REST – bzw. RESTful HTTP – als Alternative zu SOAP- und WSDL-basierten Web Services immer mehr durch, zumindest was die Open-Source- und Web-2.0-Gemeinde angeht. Wie es sich gehört, ist Rails hier ganz vorn mit dabei. Die ursprünglich im Framework enthaltene Unterstützung für SOAP ist seit Version 2.0 kein Kernbestandteil mehr und muss bei Bedarf als Plug-in nachinstalliert werden. Wie immer „opinionated" hat sich das Rails-Team dafür entschieden, REST als Standardweg sowohl für die Architektur der Web-UIs als auch für die Maschine-zu-Maschine-Kommunikation einzusetzen.

Hinter REST (REpresentational State Transfer) verbirgt sich die Architektur des World Wide Web, definiert von Miterfinder Roy T. Fielding [1]. Ganz kurz zusammengefasst lässt sie sich auf einige Grundprinzipien reduzieren: Das zentrale Element in einer REST-Architektur sind *Ressourcen*, die klar identifiziert werden können. In der konkreten Ausprägung, HTTP, sind für diese Identifikation URIs zuständig. Ressourcen werden niemals direkt angesprochen, sondern nur über *Repräsentationen*, die den Zustand der Ressource enthalten und zwischen Client und Server transferiert werden. Alle Ressourcen unterstützen die gleiche Schnittstelle, bei HTTP vor allem bestehend aus den Standardmethoden GET, PUT, POST und DELETE. Schließlich heißt das WWW nicht umsonst World Wide *Web*: Ressourcen sind miteinander vernetzt. Das spiegelt sich über Links in den Repräsentationen wider, die den Kontrollfluss des Clients und damit der Gesamtanwendung steuern können.

kurz und bündig

Inhalt

Rails bringt eine gute Unterstützung von REST und HTTP mit.
Das Standard-Scaffolding ist ein Start, aber nicht perfekt.
Mit wenig Aufwand lassen sich „RESTful" Services mit Rails erstellen.

Quellcode mit angeliefert

Nein

REST und Rails Scaffolding

Rails ist eines der wenigen populären Web-Frameworks, die den REST-Ansatz gut unterstützen. In der Datei *config.rb* bekanntgemachte Standardressourcen (Listing 1) werden auf sinnvolle URIs abgebildet und die Methoden korrekt interpretiert (Tabelle 1). Seit Rails 1.2 gibt es optional ein Scaffolding, das weitgehend RESTful ist. Dieses Scaffolding ist seit Version 2.0 sogar standardmäßig eingebaut. Das ist schon ein guter Startpunkt, aber auch nicht mehr. Damit sich eine Anwendung berechtigt mit dem Prädikat „RESTful" schmücken kann, ist Nacharbeit notwendig. Die Hauptkritikpunkte am Standard-Scaffolding von Rails sind folgende:

- *Fehlende Links*: Der Scaffold-Generator erzeugt zwar saubere URIs für die einzelnen Ressourcen, verknüpft diese aber nur in der HTML-Darstellung miteinander und auch dort nur unzureichend. Die Repräsentationen im XML-Format enthalten nur die reinen Daten. Das führt dazu, dass sich die meisten Clients einer solchen Anwendung auf eine spezifische URI-Struktur verlassen. Ein Beispiel: Man erwartet, dass man die ID eines Objekts an die URI der Collection, in der dieses Objekt enthalten ist, anhängen kann. Solche URIs sind zwar typisch und auch durchaus elegant, Clients sollten aber URIs nie konstruieren, sondern immer nur Links folgen. Man könnte sich zwar vorstellen, dass automatisch Links generiert werden, die den Assoziationen in den Modellen entsprechen, aktuell ist eine solche Unterstützung jedoch nicht vorhanden.

- *CRUD als zentrale Abstraktion*: Das Scaffolding ist explizit nur als Startpunkt gedacht und in seiner Mächtigkeit naturgemäß beschränkt. Häufig wird der generierte Code aber als Vorlage verwendet, was dazu führen kann, dass man REST mit CRUD (Create, Read, Update, Delete) gleichsetzt. Dass das clientseitige ActiveResource-Framework ebenfalls diesem Ansatz folgt, verstärkt diesen Eindruck. Tatsächlich will man aber in den seltensten Fällen direkt die Datenbankentitäten bzw. -modelle veröffentlichen; diese sind ein Implementierungsdetail. Stattdessen sollte eine explizite Schnittstellenschicht entworfen werden, die sich von der darunter liegenden Persistenzschicht durchaus deutlich unterscheiden kann.
- *Fragwürdige XML-Formate*: REST bzw. RESTful HTTP ist unabhängig vom eingesetzten Format und nicht auf XML beschränkt. Dennoch tauschen die meisten RESTful Web Services Daten über XML aus. Die durch das Scaffolding erzeugte Implementierung produziert eine XML-Darstellung, die suboptimal ist. Sie enthält Informationen über die Datentypen (die, wenn überhaupt, eigentlich in ein externes Schema gehören), verwendet keine Namespaces und ändert sich bei jeder Modelländerung, da einfach nur alle Attribute serialisiert werden.

REST.next_level!

Glücklicherweise lässt sich Rails-typisch mit wenig Code ein Web Service entwickeln, der das Attribut „RESTful" verdient – und genau das möchte ich in den nächsten Abschnitten näher erläutern. Dabei lege ich den Fokus auf die Serverseite (oder den „Provider", für die SOA-Fans). Für unser Experiment legen wir zwei einfache Modelle an: Kunde mit Name, Ort und Land sowie Kundenkontakt (ein Ansprechpartner beim Kunden) mit Vor- und Nachname und einer 1:n-Beziehung dazwischen. In *config/routes.rb* definieren wir die Ansprechpartner als Unterressourcen der Kunden (Listing 2). Da wir nur ein Modell angelegt haben und uns der Controller fehlt, gehen unsere Aktionen im Moment ins Leere. Wir verzichten bewusst auf das Scaffolding und erstellen unsere Controller explizit mit den folgenden Generatoraufrufen:

Listing 1

```
ActionController::Routing::Routes.draw do |map|
  map.resources :customers do |customers|
    customers.resources :people
  end
end
```

Listing 2

```
def index
  @customers = Customer.paginate :page => params[:page]
  respond_to do |format|
    format.xml  { response.content_type = 'application/vnd.innoq-customer+xml' }
    format.json { render :json => @customers }
  end
end
```

Listing 3

```
xml.instruct! :xml, :encoding => „UTF-8"
xml.customers :xmlns => 'http://innoq.com/schemas/customers', 'xml:base' =>
base_uri do
  xml.target! << xml_pagination(@customers)
  @customers.each { |c|
    xml.customer {
      xml.name c.name
      xml.link :rel => 'detail', :href => customer_path(c)
    }
  }
end
```

Listing 4

```
<?xml version="1.0" encoding="UTF-8"?>
<customers xmlns="http://innoq.com/schemas/customers"
          xml:base="http://localhost:3000">
  <link rel="self" href="/customers?page=2"/>
  <link rel="next" href="/customers?page=3"/>
  <link rel="prev" href="/customers?page=1"/>
  <customer>
    <name>innoQ Deutschland GmbH</name>
    <link rel="detail" href="/customers/31"/>
  </customer>
  <customer>
    <name>MegaBank Ltd.</name>
    <link rel="detail" href="/customers/32"/>
  </customer>
  ...
</customers>
```

Listing 5

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="http://innoq.com/schemas/customers">
  <link rel="self" href="http://localhost:3000/customers/1"/>
  <link rel="people" href="http://localhost:3000/customers/1/people"/>
  <name>innoQ Deutschland GmbH</name>
  <city>Ratingen</city>
  <country>Germany</country>
  <people>
    <person>
      <last_name>Meier</last_name>
      <given_name>Hans</given_name>
      <link rel="self" href="http://localhost:3000/customers/1/people/1"/>
    </person>
    <person>
      <last_name>M&#252;ller</last_name>
      <given_name>Klaus</given_name>
      <link rel="self" href="http://localhost:3000/customers/1/people/2"/>
    </person>
    <person>
      <last_name>Schulze</last_name>
      <given_name>Fritz</given_name>
      <link rel="self" href="http://localhost:3000/customers/1/people/3"/>
    </person>
  </people>
</customer>
```

```
script/generate controller Customers index update create delete
script/generate controller People index update create delete
```

Natürlich kann man auch mit dem Scaffolding beginnen und die generierten Methoden entsprechend modifizieren. Jedoch möchte ich an dieser Stelle explizit einen reinen Web Service ohne Benutzeroberfläche gestalten. Falls Sie schon Rails 2.2 verwenden, sollten Sie in *config/environment.rb* die korrekte Auswertung von Content Type- und Accept Header wieder einschalten (in den Versionen davor ist sie standardmäßig aktiviert):

```
config.action_controller.use_accept_header = true
```

Als Erstes implementieren wir im generierten *CustomersController* die *index*-Aktion (Listing 3). Diese unterscheidet sich in einigen wesentlichen Punkten vom generierten Scaffolding. Zunächst setzen wir einen spezifischen Content-Typ für „unser" XML-Format: Aus *application/xml* kann man nur schließen, dass man eine XML-Nachricht empfängt, nicht jedoch, um welchen Dokumenttyp es sich handelt. Hat man keinen passenden Standardtyp zu Hand, darf man eigene erfinden, muss diese allerdings mit dem Präfix *vnd* versehen.

Der Pagination-Mechanismus sorgt dafür, 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). In einer Weboberfläche sind wir es gewohnt, Suchergebnisse seitenweise präsentiert zu bekommen. Das ist in einem Web Service nicht anders: Auch hier möchte man sich gegen einen – absichtlichen oder versehentlichen – Zugriff auf sämtliche Kundendaten absichern. In einem Web Service im REST-Stil benutzen wir dazu das gleiche Verfahren, in diesem Fall umgesetzt durch das *will_paginate*-Plug-in [2].

Unter anderem deswegen müssen wir auch die XML-Darstellung anpassen. Das Standard-Scaffolding wandelt die Liste hierzu einfach generisch in XML um. Für die Verwendung aus dem Standard-Rails-Client, ActiveResource, ist das in Ordnung. Das dabei generierte XML genügt aber nur sehr spartanischen Ansprüchen. Stattdessen sorgen wir mit der leeren *format.xml*-Anweisung dafür, dass das Framework nach einer passenden View sucht. Diese realisieren wir mit dem XML-Builder, d.h. in einer Datei mit dem Namen *index.xml.builder* (Listing 4).

Die XML-Builder-Syntax macht es recht einfach zu erkennen, wie eine Antwort aussieht. Noch bevor die eigentlichen Inhalte dargestellt werden, werden die notwendigen Links für das Vor- und Zurückblättern im Resultat generiert. Dazu muss man eine eigene Renderer-Klasse erstellen und dem *will_paginate*-Plug-in übergeben (die Details dazu würden hier zu weit führen, Sie können den vollständigen Code aber unter [3] herunterladen). Dann folgt die Liste mit den eigentlichen Kundendaten.

Als Faustregel gilt, dass man kaum genug Links auf weiterführende Ressourcen in die Repräsentation aufnehmen kann. In der Beispielantwort sehen wir diverse Fälle. Die URI, die das Ergebnis selbst identifiziert, Links für die nächste und, falls vorhanden, die vorherige Seite und eine Verknüpfung für die einzelnen Kunden.

Folgen wir einem dieser Kundenlinks, erhalten wir eine Darstellung des Kunden in XML-Form (Listing 5). Darin sind nicht nur Daten über den Kunden, sondern auch über die assoziierten Personen enthalten, um die Anzahl der notwendigen Requests zu minimieren.

Caching

Setzt man HTTP-Requests mit dem Browser ab, sieht man nur die in der Antwort enthaltenen Daten, nicht jedoch die HTTP-Header. Abhilfe schaffen geeignete Browser-Plug-ins oder – meine Präferenz – das Schweizer Messer des HTTP-Entwicklers, das Kommandozeilenwerkzeug *curl* [4]. Bei einem lokal gestarteten Server können wir mit folgender Kommandozeile nicht nur den

Tabelle 1: Standard-Routen für Scaffolding

Routenname	HTTP-Verb	Pfad	Controller	Action
customers	GET	/customers	customers	index
formatted_customers	GET	/customers.:format	customers	index
	POST	/customers	customers	create
new_customer	GET	/customers/new	customers	new
edit_customer	GET	/customers/:id/edit	customers	edit
customer	GET	/customers/:id	customers	show
	PUT	/customers/:id	customers	update
	DELETE	/customers/:id	customers	destroy
customer_people	GET	/customers/:customer_id/people	people	index
	POST	/customers/:customer_id/people	people	index
new_customer_person	GET	/customers/:customer_id/people/new	people	create
edit_customer_person	GET	/customers/:customer_id/people/:id/edit	people	new
customer_person	GET	/customers/:customer_id/people/:id	people	edit
	PUT	/customers/:customer_id/people/:id	people	show
	DELETE	/customers/:customer_id/people/:id	people	update

eigentlichen Inhalt der Antwort, sondern auch die Header anzeigen lassen:

```
curl -i http://localhost:3000/customers/
```

Dabei werden unter anderem folgende Headerinformationen ausgegeben:

```
HTTP/1.1 200 OK
Cache-Control: private, max-age=0, must-revalidate
Date: Sun, 16 Nov 2008 12:25:07 GMT
Content-Type: application/vnd.innoq-customer+xml; charset=utf-8
Etag: "cfbbdde3db0a7e28885c31dc14e076ac"
```

Listing 6

```
def create
  @customer = Customer.find(params[:customer_id])
  @person = Person.new_from_xml(params[:person])
  @customer.people << @person
  respond_to do |format|
    if @person.save
      format.xml  { response.content_type = 'application/vnd.innoq-customer+xml'
                render :action => 'show', :status => :created,
                :location => customer_person_url(@customer, @person) }
    else
      format.xml  { render :xml => @person.errors,
                    :status => :unprocessable_entity }
    end
  end
end
```

Listing 7

```
def update
  @customer = Customer.
                      find(params[:id])
  if precondition_met?(@customer)
    respond_to do |format|
      if @customer.update_attributes_from_xml(params[:customer])
        format.xml  {
          response.content_type = 'application/vnd.innoq-customer+xml'
          render :action => 'show'
      }
      else
        format.xml  { render :xml => @customer.errors,
                      :status => :unprocessable_entity }
      end
    end
  else
    head :precondition_failed
  end
end

def precondition_met?(obj)
  response.etag = obj
  if_match_header = request.headers['IF-MATCH']
  if_match_header.nil? || (if_match_header == response.etag)
end
```

Neben dem *Content-Type*, den wir in der *index*-Methode explizit gesetzt haben, sind die *Cache-Control*-Header, das Datum und das *Etag* interessant. Legt man nichts Anderes fest, wird dem Client mit den Parametern *max-age=0* und *must-revalidate* mitgeteilt, dass er die Informationen nicht (bzw. nur auf eigene Gefahr) zwischenspeichern darf. Wir können die Anzahl der potenziellen Anfragen dramatisch reduzieren, wenn wir dem Client das Caching erlauben. Rails unterstützt dies elegant mit der Controller-Methode *expires_in*:

```
expires_in 5.seconds
```

Nun informiert der Server den Client bei einer Anfrage, dass die Daten mindestens fünf Sekunden lang gültig sind und sich eine Anfrage vorher nicht lohnt:

```
Cache-Control: private, max-age=5
```

Was soll der Client nun tun, wenn seine lokale Kopie der Daten älter als fünf Sekunden ist? Er könnte die Repräsentation der Ressource einfach noch einmal komplett herunterladen. Klüger ist es, eine bedingte Abfrage, ein *conditional GET*, durchzuführen. Mit *curl* können wir Header über die Option *-H* setzen:

```
curl -i http://localhost:3000/customers/ -H 'If-Modified-Since: Sun, 16 Nov 2008 12:50:19 GMT'
```

Als Ergebnis erhalten wir eine Antwort mit dem HTTP-Statuscode *304* (*Not Modified*, unverändert) und sparen es uns, die Inhalte noch einmal zu übertragen. Damit das funktioniert, muss der Controller den *If-Modified-Since*-Header auswerten:

```
response.last_modified = Customer.maximum('updated_at').utc
  if request.fresh?(response)
      head :not_modified
  else
    ... # normale Verarbeitung
```

Neben dem Datums-Header in Verbindung mit *If-Modified-Since* gibt es noch das bereits erwähnte *Etag*, ein für jeden Zustand der Ressource unterschiedlichen Wert. Auch dieser kann für ein *Conditional GET* verwendet werden, der in diesem Fall zu einem *304* führt:

```
curl -i http://localhost:3000/customers/ -H 'If-None-Match:
"cfbbdde3db0a7e28885c31dc14e076ac"'
```

Rails berechnet das *Etag* automatisch. Das funktioniert zwar, ist aber nicht sonderlich effizient. Die gesamte Antwort muss berechnet, dann der Hash-Wert ermittelt und schließlich mit dem vom Client gesendeten verglichen werden – was dazu führen kann, dass die Antwort wieder verworfen und nur *304* zurückgegeben wird. Alternativ kann das *Etag* gesetzt werden, in dem man der Funktion *etag=* ein oder mehrere Objekte übergibt:

```
response.etag = @customers
```

In Rails 2.2 gibt es eine neue, praktische Methode namens *stale?*, die das Setzen und Prüfen der Header vereint:

```
if stale?(:last_modified => Customer.maximum('updated_at').utc, :etag => @customers)
... # normale Request-Verarbeitung
End
```

Ein *else*-Zweig ist dabei nicht nötig, das Framework versendet den entsprechenden Code. Caching und *Conditional GET* sind die zentrale Voraussetzung für die Skalierbarkeit der Webarchitektur: Nichts ist effizienter als Anfragen, die gar nicht erst gestellt werden müssen.

Bearbeitung von Ressourcen

Für das Anlegen neuer Ressourcen kann entweder *POST* oder *PUT* eingesetzt werden. Der Unterschied besteht darin, wer die URI des neu anzulegenden Objekts bestimmt. Ein *PUT* „wirkt" gemäß HTTP-Spezifikation auf die Ressource, deren URI verwendet wird, d.h. sie wird entweder unter der vom Client vorgegebenen URI angelegt oder aktualisiert, falls sie schon vorhanden ist. Ein *POST* erzeugt eine neue Ressource, bei der der Server für die Vergabe der neuen URI zuständig ist. Das ist in der Regel besser und auch der Weg, der beim Rails-Ressource-Mapping verwendet wird. Als Beispiel sehen wir uns das Anlegen neuer Ansprechpartner bei einem Kunden an. Ein Client findet zur Laufzeit über den Link mit dem Attribut *people* den URI heraus, den er dazu verwenden kann. Durch die Konfiguration in der Datei *routes.rb* wird ein *POST* zu */customers/:customer_id/people* auf die Methode *create* im *PeopleController* abgebildet (Listing 6, die Methode *new_from_xml* in der Klasse *Person* ist aus Platzgründen nicht dargestellt). Wir verwenden wieder *curl*, um die Daten für eine neue Person zum Server zu senden. Mit dem Schalter *-d* können wir Daten übergeben, die per *POST* übermittelt werden sollen:

```
curl -i http://localhost:3000/customers/1/people \
-H 'Content-type: application/xml' -d '<?xml version="1.0" encoding="UTF-8"?>
<person xmlns="http://innoq.com/schemas/customers">
  <last_name>Kunze</last_name>
  <given_name>Reinhard</given_name>
</person>'
```

Als Antwort erhalten wir den passenden HTTP-Statuscode *204 Created,* der URI der neu angelegten Person wird in einem Location-Header gesetzt. Für den Fall, dass der Server beim Anlegen etwas ergänzt hat (in unserem Fall ein *link*-Element), wird eine Repräsentation der Ressource mitgeliefert:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.innoq-customer+xml; charset=utf-8
Content-Length: 240
Location: http://localhost:3000/customers/1/people/10

<?xml version="1.0" encoding="UTF-8"?>
<person xmlns="http://innoq.com/schemas/customers">
  <last_name>Kunze</last_name>
  <given_name>Reinhard</given_name>
  <link rel="self" href="http://localhost:3000/customers/1/people/10"/>
</person>
```

Sehen wir uns zum Abschluss das Aktualisieren eines bestehenden Kunden an. Interessant ist, dass wir auch hier die *Etag*- und Datums-Header verwenden können.

Ein Standardproblem bei verteilten Anwendungen ist der konkurrierende Zugriff durch unterschiedliche Clients. Nehmen wir an, zwei Clients versuchen nacheinander einen Kunden zu aktualisieren. Wie verhindern wir, dass ein Client die Änderungen eines anderen unwissentlich überschreibt? Das HTTP-Protokoll bietet hierzu den *If-Match*-Header an. Der Client kann damit dem Server eine Vorbedingung mitteilen, die erfüllt sein muss, damit der Request ausgeführt wird: Das aktuelle *Etag* der Ressource muss mit dem übermittelten übereinstimmen. Führt also Client 1 ein *GET* auf eine Ressource aus und bekommt das *Etag A*, gefolgt von Client 2, der das *Etag B* erhält, senden beide bei darauf folgenden *PUT*-Requests das *Etag* in einem *If-Match*-Header mit. Hier das Beispiel für die Aktualisierung eines Kunden, für den wir vorher das *Etag 2ee-280e77ea055825f6e78fb55f5401a* erhalten haben:

```
curl -i http://localhost:3000/customers/1 -H 'If-Match:
"2ee280e77ea055825f6e78fb55f5401a"' -H 'Content-type: application/xml' -X PUT -d
'<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="http://innoq.com/schemas/customers">
  <name>innoQ Deutschland GmbH</name>
  <city>Langenfeld</city>
  <country>Germany</country>
</customer>'
```

Dieser Request würde erfolgreich durchgeführt, weil das *Etag* mit dem *If-Match*-Header übereinstimmt. Ist dies nicht der Fall, antwortet der Server mit einem *412 Precondition Failed*. Das funktioniert in der aktuellen Rails-Version nicht von allein, lässt sich aber mit wenigen Zeilen Code ergänzen (Listing 7). Dies zeigt, wie optimistische Nebenläufigkeitskontrolle mit reinen HTTP-Mitteln interoperabel umgesetzt werden kann.

Fazit

Rails unterstützt auch erweiterte HTTP-Funktionen, und die generelle Philosophie von Rails passt gut zum REST-Ansatz. Die vom Scaffolding generierten Controller sind dabei ein guter Startpunkt, aber noch nicht wirklich „RESTful". Nimmt man etwas Mehraufwand in Kauf, eignet sich Rails jedoch hervorragend zum Entwickeln von Web Services, die den REST-Prinzipien folgen. Damit profitiert REST von Rails und umgekehrt: Hat man sich für REST entschieden, passt Rails besonders gut; setzt man ohnehin schon Rails ein, ist REST eine naheliegende Wahl.

Links und Literatur

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[2] http://github.com/mislav/will_paginate/tree/master

[3] http://curl.haxx.se/

[4] http://github.com/stilkov/restful-rails-samples/tree/master

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, wo er sich vorwiegend mit serviceorientierten Architekturen auf Basis von Web Services und REST beschäftigt.