

Tobias Günther

REST in Rails

Teil 2: Die Praxis der REST-Architektur in Ruby On Rails

Rails 1.2 bringt als große Neuerung die Unterstützung der REST-Architektur mit sich. Nachdem sich Teil 1 (siehe T3N Magazin Nr. 7) um die theoretischen Hintergründe von REST drehte, geht es nun ans Eingemachte: Eine kleine Artikel-Verwaltung soll RESTful umgesetzt werden.

Die REST-basierte Entwicklung in Rails bietet nur wenige Unterschiede zur „klassischen“, altbekannten Vorgehensweise. Neu sind vor allem folgende Punkte:

- `respond_to` im Controller ermöglicht es, mit wenigen Handgriffen unterschiedliche Response-Formate auszugeben.
- Das Routing erhält mit der Methode `resources` eine REST-Implementierung.
- `Redirects` im Controller und `Links` in den Views verfügen über neue Methoden und Helper.

Über diese größeren Neuerungen hinaus bleibt uns aber größerer Lernaufwand erspart – die meisten anderen Bereiche werden durch REST nicht berührt. Daher werden in diesem Artikel auch nur die entscheidenden, neuen Bereiche angesprochen.

Auf die Plätze... fertig... coden!

Wie bei jedem Rails-Projekt sind auch hier die ersten Schritte das Generieren des Anwendungs-Codes über „rails projektname“ und das Anlegen der entsprechenden Datenbank(en).

Interessant wird es, wenn wir nun den neuen Script-Generator „`scaffold_resource`“ einsetzen, um uns ein Grundgerüst bestehend aus Model, Controller und allem was dazu gehört erzeugen zu lassen. Der Generator erwartet als ersten Parameter den Namen des Models. Aus dem Namen des Models generiert Rails automatisch dann auch den entsprechenden Controller: Der Plural des Models wird hierfür als Name verwendet.

Weitere Parameter erlauben es uns, in der zugehörigen Datenbank-Migration gleich bestimmte Felder anlegen zu lassen und in der View ebenfalls diese Felder zu nutzen. In unserem Beispiel legen wir das Model „`article`“ mitsamt den Tabellenfeldern `title`, `category`, `teaser`, `keywords`, `date` und `content` sowie den Controller „`articles_controller`“ an. Dazu genügt folgender Befehl:

```
Ruby On Rails
```

```
$ ruby script/generate scaffold_resource article title:string teaser:text keywords:string date:date content:text
```

Listing 1

Mit „`rake db:migrate`“ wird nun die Datenbanktabelle auch tatsächlich inklusive der gewünschten Felder angelegt. Ein Blick in den generierten Code gibt uns zwei interessante Auskünfte:

1. Im Hinblick auf die Models bringt der REST-Ansatz nichts Neues mit sich.
2. Im Hinblick auf den Controller allerdings schon (hierzu später mehr)

Starten wir doch kurz unseren Server (ab Rails 1.2 übrigens standardmäßig Mongrel und nicht mehr WEBrick):

```
Ruby On Rails
```

```
$ ruby script/server
```

Listing 2

Nun schicken wir unseren Browser auf die Adresse `http://localhost:3000/articles` – Sie werden sehen, dass sich alles anfühlt, wie bei einer Rails-Anwendung ohne REST auch. Alles – bis auf die URLs und den dahinterliegenden Code.

Erinnern Sie sich: Bei REST wird die Action nun über das verwendete HTTP-Verb bestimmt und kommt daher nicht mehr in der URL vor. Bis auf die Ausnahmen `new` und `edit` enthalten unsere CRUD-URLs jetzt also nur noch den Controller und die ID des Datensatzes und sehen damit alle folgendermaßen aus:

URL `http://localhost:3000/articles/1`

Um nun zu beurteilen, was durch diesen Aufruf passiert, reicht ein Blick auf die URL nicht aus. Erst die verwendete HTTP-Methode gibt Auskunft darüber, was mit der Ressource geschieht:

Aktion	URL	HTTP-Methode	Link-Helper
index	/articles	GET	articles_path
show	/articles/1	GET	article_path(1)
new	/articles/new	GET	new_article_path
edit	/articles/1/edit	GET	edit_article_path(1)
create	/articles	POST	articles_path
update	/articles/1	PUT	article_path(1)
destroy	/articles/1	DELETE	article_path(1)

Kreative Namensgeber werden nun vielleicht enttäuscht sein: keine abwechslungsreichen URLs wie `/articles/add_new_article` oder `/articles/delete_article/1` mehr. Dafür werden Sie aber mit einem einheitlichen Set an URL-Standards belohnt, das von Applikation zu Applikation konstant bleibt.

Ein Problem gilt es noch zu lösen (das übernimmt allerdings netterweise Rails für uns): Aktuelle Browser unterstützen nur die klassischen HTTP-Methoden GET und POST. Daher muss das Framework hier etwas nachhelfen, um eine massenkompatible Lösung zu schaffen.

Diese Lösung sieht folgendermaßen aus: Auch Rails verwendet hinter den Kulissen lediglich die beiden Klassiker GET und POST. Die Helper-Methoden, die die neuen REST-Links erstellen, simulieren die HTTP-Verben PUT und DELETE über versteckte Formularfelder. Ein Link zum Löschen einer Ressource wird von Rails so mit JavaScript versehen, dass beim Absenden daraus ein über POST versendetes Formular wird, welches ein hidden field mit dem Namen „`_method`“ und dem Wert „`delete`“ enthält. Kommt dieser Request bei unserer Rails-Anwendung an, weiß sie dadurch automatisch, dass es sich „eigentlich“ um einen DELETE-Request handeln soll.

Was hat sich nun im Code des Controllers verändert? Vor allem die Möglichkeit, sehr einfach unterschiedliche Ausgabeformate liefern zu können, macht REST so attraktiv. Über den `respond_to`-Block ist dies schnell und im Idealfall ohne duplizierten Code realisierbar:

```
Ruby On Rails
```

```
# GET /articles/1
# GET /articles/1.xml
```

```
def show
  @article = Article.find(params[:id])
  respond_to do |format|
    format.html # show.rhtml
    format.xml { render :xml => @article.to_xml }
  end
end
```

Listing 3

Die Entscheidung, welches Format nun tatsächlich an den Client zurückgegeben wird, kann über zwei verschiedene Wege getroffen werden: Entweder das Accept-Feld des HTTP-Headers gibt vor, welche Art Content der Client wünscht oder die URL des Requests enthält das Format explizit (z. B. durch /articles.xml). Um unsere Anwendung nun zum Beispiel auch JSON ausgeben zu lassen, brauchen wir nur folgende Zeile in den respond_to-Block der show-Methode einzufügen:

```
Ruby On Rails
respond_to-Block in app/controllers/articles_controller.rb
format.json { render :json => @article.to_json }
```

Listing 4

link_to & Co.

Im View-Code gibt es eine weitere Neuheit: link_to wurde mit einigen REST-spezifischen Helper-Methoden aufpoliert:

```
link_to in Views, z. B. app/views/articles/index.rhtml
<%= link_to 'Show', article_path(article) %> # /articles/1
<%= link_to 'Edit', edit_article_path(article) %> # /articles/1/edit
<%= link_to 'Destroy', article_path(article), :confirm => 'Are you sure?', :method => :delete %>
# /articles/1 - allerdings mit JavaScript-Zusätzen, um die HTTP-Methode DELETE mitzugeben
```

Listing 5

Anstatt wie in klassischen Applikationen :controller-, :action- und :id-Parameter an link_to zu übergeben, können jetzt die *_path-Methoden verwendet werden. Im obigen Code-Beispiel wird auch deutlich, wie die PUT und DELETE-Requests zustande kommen – durch den Parameter :method nämlich.

Insgesamt sind es sieben neue Standard-Path-Methoden, die Rails für uns erstellt hat – eine für jede der sieben REST-Aktionen. In der obigen Tabelle sind sie neben der jeweiligen Aktion aufgeführt.

Mit den *_path-Methoden werden gleichzeitig auch neue Helper-Methoden für die Verwendung im Controller generiert. Sie sind identisch zu den _path-Methoden, außer dass sie auf _url enden (z. B. articles_url statt articles_path). Diese Helper werden klassischerweise dann verwendet, wenn es darum geht, der redirect_to-Methode ein Ziel zu geben.

```
app/controllers/articles_controller.rb
format.html { redirect_to article_url(@article) }
```

Listing 6

Neues Routing

Doch wo kommen diese scheinbar von Geisterhand erzeugten neuen Helper-Methoden überhaupt her? Das Routing von REST-Applikationen ist ebenfalls neu und generiert uns netterweise all diese _path- und _url-Methoden:

```
config/routes.rb
ActionController::Routing::Routes.draw do |map|
  map.resources :articles
  ....
end
```

Listing 7

Eigene Actions und Helpers anlegen

Die REST-Implementierung in Rails unterstützt standardmäßig die klassischen CRUD-Actions (Create, Read, Update, Delete). Doch in den meisten Anwendungen müssen darüber hinaus weitere Funktionen verfügbar sein. Diese müssen einerseits als Actions im Controller angelegt werden, andererseits muss aber auch das Routing auf diese neuen, für Rails unbekanntenen Actions eingestellt werden, damit die Applikation weiß, wie sie auf eine bisher unbekannte URL reagieren soll.

In config/routes.rb muss daher ein neuer Eintrag für jede neue Action angelegt werden:

```
config/routes.rb
map.resources :articles, :member => { :bsp_action => :get }
```

Listing 8

Der :member-Parameter erwartet einen Hash, der Rails sagt, welche Action über welche HTTP-Methode (:get, :post, :delete, :put oder :any) erwartet wird. Außer :member kann auch :collection (für Actions, die statt mit einer einzigen Ressource mit mehreren arbeiten) und :new (Actions, die neue Ressourcen anlegen) zum Einsatz kommen.

Dieser neue map.resources-Eintrag generiert dann die erforderlichen Routes und Helper-Methoden:

Action-Name	URL	Helper-Methode	HTTP
bsp_action	/articles;bsp_action	bsp_action_articles_url	GET

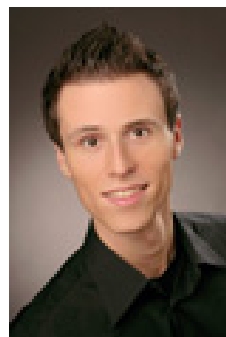
Wird dann beispielsweise dieser Request über POST abgesendet, erhalten wir einen Routing-Error. Denn in unserer routes.rb haben wir definiert, dass die Action nur über GET aufgerufen werden soll.

Ausblick auf Rails 2.0

Rails 2.0 wird eine kleine Änderung in den Naming Conventions bei den REST-URLs mit sich bringen. Die Verwendung des Semikolons, wie zum Beispiel in „article/1/edit“, wird zugunsten der weniger gewöhnungsbedürftigen Schreibweise „article/1/edit“ aufgegeben. Hartcodierte Links sollten also am besten nicht mehr das Semikolon nutzen, da map.resources dies in Rails 2.0 nicht mehr unterstützen wird.

 [Softlink 1500](#)

DER AUTOR



Tobias Günther leitet seit drei Jahren die Web-Agentur puremedia in Stuttgart (www.puremedia-online.de). Schwerpunkte seiner Arbeit sind die Themen AJAX / JavaScript und Ruby on Rails.