RESTful Rails Development

Ralf Wirdemann und Thomas Baustert
www.b-simple.de
Hamburg

31. Januar 2007

This work is licensed under the Creative Commons Attribution-No Derivative Works 2.0 Germany License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Ralf Wirdemann, Hamburg im Januar 2007

Inhaltsverzeichnis

1 RESTful Rails . . . 1

1.1 Was ist REST? . . . 2
1.2 Warum REST? . . . 3
1.3 Was ist neu? . . . 3
1.4 Vorbereitung . . . 4
1.4.1 Rails 1.2 . . . 4
1.5 Resource Scaffolding . . . 4
1.6 Das Modell . . . 5
1.7 Der Controller . . . 5
1.7.1 REST-URLs . . . 6
1.7.2 REST-Actions verwenden respond_to . . . 7
1.7.3 Accept-Feld im HTTP-Header . . . 8
1.7.4 Formatangabe via URL . . . 9
1.8 REST-URLs in Views . . . 10
1.8.1 New und Edit . . . 11
1.8.2 Path-Methoden in Formularen: Create und Update . . . 12
1.8.3 Destroy . . . 13
1.9 URL-Methoden im Controller . . . 13
1.10 REST-Routing . . . 14
1.10.1 Konventionen . . . 15
1.10.2 Customizing . . . 15
1.11 Verschachtelte Ressourcen . . . 16
1.11.1 Anpassung des Controllers . . . 18
1.11.2 Neue Parameter für Path- und URL-Helper . . . 19
1.11.3 Zufügen neuer Iterationen . . . 20
1.11.4 Bearbeiten existierender Iterationen . . . 22
1.12 Eigene Actions . . . 23
1.12.1 Sind wir noch DRY? . . . 26

1.13 Eigene Formate 26
1.14 RESTful AJAX 27
1.15 Testen 28
1.16 RESTful Clients: ActiveResource 29
1.17 Abschliessend 31

Literaturverzeichnis **33**

Kapitel 1

RESTful Rails

HTTP kennt mehr als GET und POST. Eine Tatsache, die bei vielen Web-Entwicklern in Vergessenheit geraten ist. Allerdings ist das auch nicht besonders verwunderlich, wenn man bedenkt, dass Browser sowieso nur GET- und POST-Aufrufe unterstützen.

GET und POST sind HTTP-Methoden, die als Teil des HTTP-Requests vom Client an den Server übertragen werden. Neben GET und POST kennt HTTP die Methoden PUT und DELETE, die dem Server die Neuanlage bzw. das Löschen einer Web-Ressource anzeigen sollen.

In diesem Tutorial geht es um die Erweiterung des Sprachrepertoires von uns Web-Entwicklern um die HTTP-Methoden PUT und DELETE. Die Verwendung von PUT und DELETE neben GET und POST wird unter dem Begriff REST subsumiert, für das Rails seit Version 1.2 Unterstützung bietet.

Das Tutorial beginnt mit einer kurzen Einführung in die Hintergründe und Konzepte von REST. Darauf aufbauend werden einige Gründe für die REST-basierte Entwicklung von Rails-Anwendungen genannt. Es folgt die detaillierte Einführung des technischen REST-Handwerkszeugs am Beispiel eines per Scaffolding generierten REST-Controllers und des zugehörigen Modells. Aufbauend auf dieser technischen Grundlage erläutert das Kapitel anschliessend die Funktionsweise und Anpassung des zugrunde liegenden REST-Routings. Im Abschnitt *Verschachtelte Ressourcen* führt das Tutorial in die hohe Schule der REST-Entwicklung ein und erklärt, wie Ressourcen im Sinne einer Eltern-Kind-Beziehung verschachtelt werden, ohne dabei das Konzept von REST-URLs zu verletzen. Das Tutorial schliesst mit je einem Abschnitt über REST und AJAX und das Schreiben von Tests für REST-Controller sowie einer Einführung in ActiveResource, dem clientseitigen Teil von REST.

Bevor es los geht, ein letzter Hinweis: Dieses Tutorial setzt Grundkenntnisse der Rails-Entwicklung voraus. Sofern diese noch nicht vorhanden sind, sollten Sie zunächst ein entsprechendes Rails-Tutorial durcharbeiten. Gute Tutorials finden Sie im Internet (siehe [3] und [4] oder auch [5]) oder in entsprechenden Büchern (siehe [1] und [2]).

1.1 Was ist REST?

Der Begriff REST stammt aus der Dissertation von Roy Fielding [6] und steht für *Representational State Transfer*. REST beschreibt ein Architektur-Paradigma für Web-Anwendungen, bei dem Web-Ressourcen mittels der Standard HTTP-Methoden GET, POST, PUT und DELETE angefordert und manipuliert werden.

Eine Ressource im Sinne von REST ist eine URL-adressierbare Entität, mit der über HTTP interagiert werden kann. Ressourcen werden in unterschiedlichen Formaten repräsentiert (zum Beispiel HTML, XML oder RSS), je nachdem, was sich der Client wünscht. Ressourcen-URLs sind eindeutig. Eine Ressource-URL adressiert damit nicht, wie in traditionellen[1] Rails-Anwendungen, ein Modell und die darauf anzuwendende Aktion, sondern nur noch die Ressource selber (siehe Abbildung 1.1).

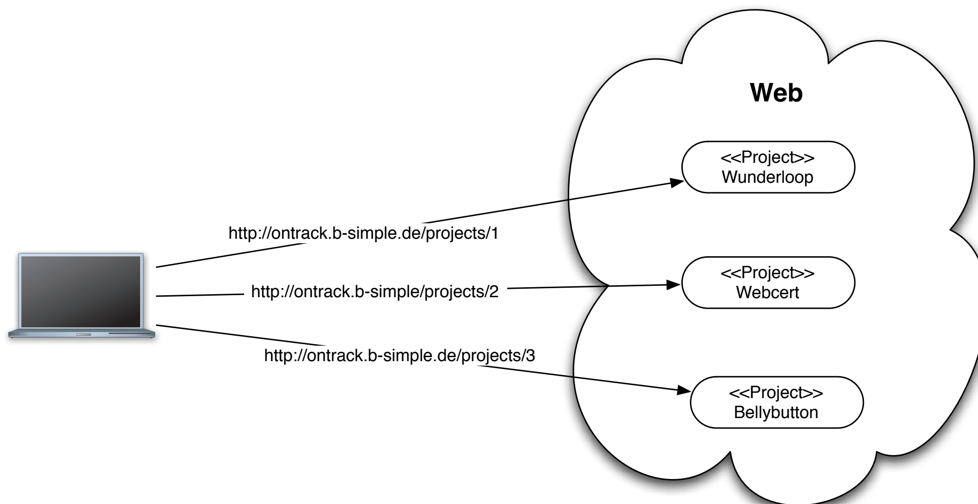

Abbildung 1.1: Ressourcen im Web und ihre URLs

Alle drei Ressourcen der Abbildung werden durch die im vorderen Teil identische URL *http://ontrack.b-simple.de/projects*, gefolgt von der eindeutigen Id der Ressource, adressiert. Die URL drückt nicht aus, was mit der Ressource geschehen soll.

Im Kontext einer Rails-Applikation ist eine Ressource die Kombination eines dedizierten Controllers und eines Modells. Rein technisch betrachtet handelt es sich bei den Project-Ressourcen aus Abbildung 1.1 also um Instanzen der ActiveRecord-Klasse *Project* in Kombination mit einem *ProjectsController*, der für die Manipulation der Instanzen zuständig ist.

[1] Wenn wir den Unterschied zwischen REST- und Nicht-REST-basierter Rails-Entwicklung deutlich machen wollen, verwenden wir dafür den Begriff *traditionell*. Traditionell steht hier nicht im Sinne von *alt* oder gar *schlecht*, sondern dient einzig und allein dem Zweck, einen Bezug zu einem äquivalenten Nicht-REST-Konzept herzustellen, um die neue Technik mit Hilfe dieses Vergleiches besser erklären zu können.

1.2 Warum REST?

Dies ist eine berechtigte Frage, insbesondere wenn man bedenkt, dass wir seit mehr als zwei Jahren erfolgreich Rails-Anwendungen auf Basis des bewährten MVC-Konzepts entwickeln. Aber: REST zeigt uns, dass Rails auch Raum für konzeptionelle Verbesserungen lässt, wie die folgende Liste von Eigenschaften REST-basierter Anwendungen verdeutlicht:

Saubere URLs. REST-URLs repräsentieren Ressourcen und keine Aktionen. URLs genügen immer dem selben Format, d.h. erst kommt der Controller und dann die Id der referenzierten Ressource. Die Art der durchzuführenden Manipulationen wird vor der URL versteckt und mit Hilfe eines HTTP-Verbs ausgedrückt.

Unterschiedliche Response-Formate. REST-Controller werden so entwickelt, dass Actions unterschiedliche Response-Formate liefern können. Je nach Anforderung des Clients liefert eine Action HTML, XML oder auch RSS. Die Anwendung wird so Multiclient-fähig.

Weniger Code. Die Entwicklung Multiclient-fähiger Actions vermeidet Wiederholungen im Sinne von DRY[2] und resultiert in Controller mit weniger Code.

CRUD-orientierte Controller. Controller und Ressource verschmelzen zu einer Einheit, so dass jeder Controller für die Manipulation von genau einem Ressourcentyp zuständig ist.

Sauberes Anwendungsdesign. REST-basierte Entwicklung resultiert in ein konzeptionell sauberes und wartbares Anwendungsdesign.

Die genannten Eigenschaften werden im weiteren Verlauf dieses Tutorials an Hand von Beispielen verdeutlicht.

1.3 Was ist neu?

Wenn Sie jetzt denken, dass REST-basiertes Anwendungsdesign alles bisher gekannte über den Haufen wirft, dann können wir Sie beruhigen: REST ist immer noch MVC-basiert und lässt sich, rein technisch gesehen, auf folgende Neuerungen reduzieren:

- Die Verwendung von *respond_to* im Controller.
- Neue Helper für Links und Formulare.
- Verwendung von URL-Methoden in Controller-Redirects.
- Neue Routen, erzeugt von der Methode *resources* in routes.rb.

Einmal verstanden und anschliessend konsequent angewandt, ergibt sich ein REST-basiertes Anwendungsdesign quasi von selbst.

[2] Don't repeat yourself

1.4 Vorbereitung

Wir beschreiben die REST-spezifischen Neuerungen in Rails am Beispiel der in unserem Buch *Rapid Web Development mit Ruby on Rails* [1] vorgestellten Projektmanagement-Anwendung *Ontrack*. Wir werden die Anwendung dabei nicht vollständig nachentwickeln, sondern verwenden nur dieselbe Terminologie, um einen fachlichen Rahmen für die REST-Konzepte zu schaffen. Los geht es mit der Generierung des Rails-Projekts:

```
> rails ontrack
```

Es folgt die Anlage der Development- und Testdatenbank:

```
> mysql -u rails -p
Enter password: *****

mysql> create database ontrack_development;
mysql> create database ontrack_test;
mysql> quit
```

1.4.1 Rails 1.2

Wir gehen davon aus, dass die wenigsten Leser Rails 1.2 als systemweite, d.h. für alle Rails-Anwendungen zu verwendende Rails-Version installieren wollen. Deshalb schlagen wir eine projektspezifische Installation von Rails 1.2 vor, d.h. Rails 1.2 wird nur vom Beispielprojekt *ontrack* verwendet. Laut Dokumentation heisst der Tag für Rails 1.2 *rel_1-2-1*. Folgendes rake-Kommando installiert diese Version für ontrack:

```
> cd ontrack
> rake rails:freeze:edge TAG=rel_1-2-1
```

Alternative Installationsvarianten bzw. Informationen über neuere Rails-Versionen finden Sie auf der Rails-Homepage *http://www.rubyonrails.org*.

1.5 Resource Scaffolding

REST-orientierte Rails-Entwicklung lässt sich am einfachsten am Beispiel einer vom neuen Scaffold-Generator *scaffold_resource* erzeugten Ressource *Project* erläutern. Der Generator erhält neben dem Ressourcennamen *project* eine optionale Liste von Modellattributen und deren Typen. Die übergebenen Attribute werden für die Erzeugung des Migrationsskriptes sowie der Felder in den generierten Views benötigt:

```
> cd ontrack
> ruby script/generate scaffold_resource project name:string desc:text
exists  app/models/
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/projects
    exists  test/functional/
```

```
    exists  test/unit/
    create  app/views/projects/index.rhtml
    create  app/views/projects/show.rhtml
    create  app/views/projects/new.rhtml
    create  app/views/projects/edit.rhtml
    create  app/views/layouts/projects.rhtml
    create  public/stylesheets/scaffold.css
    create  app/models/project.rb
    create  app/controllers/projects_controller.rb
    create  test/functional/projects_controller_test.rb
    create  app/helpers/projects_helper.rb
    create  test/unit/project_test.rb
    create  test/fixtures/projects.yml
    create  db/migrate
    create  db/migrate/001_create_projects.rb
     route  map.resources :projects
```

Der Generator erzeugt neben Modell, Controller und Views ein vorausgefülltes Migrationsskript sowie einen neuen Mapping-Eintrag *map.resources :projects* in der Routing-Datei *config/routes.rb*. Insbesondere letzterer ist für den REST-Charakter des erzeugten Controllers verantwortlich. Aber wir wollen nichts vorwegnehmen und stattdessen die erzeugten Artefakte Schritt für Schritt erklären.

1.6 Das Modell

Wie eingangs erwähnt, sind REST-Ressourcen im Kontext von Rails die Kombination aus Controller und Modell. Das Modell ist dabei eine normale ActiveRecord-Klasse, die von *ActiveRecord::Base* erbt:

```
class Project < ActiveRecord::Base
end
```

In Bezug auf das Modell gibt es somit nichts neues zu lernen. Vergessen Sie trotzdem nicht, die zugehörige Datenbanktabelle zu erzeugen:

```
> rake db:migrate
```

1.7 Der Controller

Der generierte *ProjectsController* ist ein CRUD-Controller zum Manipulieren der Ressource *Project*. Konkret bedeutet dies, dass der Controller sich genau auf einen Ressourcentyp bezieht und für jede der vier CRUD-Operationen eine Action[3] zur Verfügung stellt:

[3] Neben den vier CRUD-Actions enthält der Controller die zusätzliche Action *index* zur Anzeige einer Liste aller Ressourcen dieses Typs, sowie die beiden Actions *new* und *edit* zum Öffnen des Neuanlage- bzw. Editier-Formulars.

Listing 1.1: ontrack/app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  # GET /projects
  # GET /projects.xml
  def index...

  # GET /projects/1
  # GET /projects/1.xml
  def show...

  # GET /projects/new
  def new...

  # GET /projects/1;edit
  def edit...

  # POST /projects
  # POST /projects.xml
  def create...

  # PUT /projects/1
  # PUT /projects/1.xml
  def update...
  end

  # DELETE /projects/1
  # DELETE /projects/1.xml
  def destroy...
end
```

Im Grunde genommen also auch noch nicht viel neues: Es gibt Actions zum Erzeugen, Bearbeiten, Aktualisieren und Löschen von Projekten. Controller und Actions sehen zunächst ganz normal, d.h. wie traditionelle Controller aus. Es fällt jedoch auf, dass der Generator zu jeder Action einen Kommentar mit der zugehörigen Aufruf-URL inklusive HTTP-Verb generiert hat. Dies sind die REST-URLs, die wir im folgenden Abschnitt näher beschreiben.

1.7.1 REST-URLs

REST-URLs bestehen nicht, wie bisher üblich, aus Controller- und Action-Name sowie optionaler Modell-Id (z.B. */projects/show/1*), sondern nur noch aus Controller-Namen gefolgt von der Id der zu manipulierenden Ressource:

```
/projects/1
```

Durch den Wegfall der Action ist nicht mehr erkennbar, was mit der adressierten Ressource geschehen soll. Soll die durch obige URL adressierte Ressource angezeigt oder gelöscht werden? Antwort auf diese Frage liefert die verwendete HTTP-

Methode[4], die für den Aufruf der URL verwendet wird. Die folgende Tabelle setzt die vier HTTP-Verben zu den REST-URLs in Beziehung und verdeutlicht, welche Kombination zum Aufruf welcher Action führt:

Tabelle 1.1: Standart Path-Methoden

HTTP-Verb	REST-URL	Action	URL ohne REST
GET	/projects/1	show	GET /projects/show/1
DELETE	/projects/1	destroy	GET /projects/destroy/1
PUT	/projects/1	update	POST /projects/update/1
POST	/projects	create	POST /projects/create

Für alle Operationen bis auf POST, da bei einer Neuanlage noch keine Id existiert, sind die URLs identisch. Allein das HTTP-Verb entscheidet über die durchzuführende Aktion. URLs werden DRY und adressieren Ressourcen statt Aktionen.

Hinweis: Die Eingabe der URL *http://localhost:3000/projects/1* im Browser ruft immer *show* auf. Browser unterstützen weder PUT noch DELETE. Für die Erzeugung eines Links zum Löschen einer Ressource stellt Rails einen speziellen Helfer bereit, der das HTTP-Verb DELETE per Post in einem Hidden-Field an den Server übermittelt (siehe Abschnitt 1.8.3). Gleiches gilt für PUT-Requests zur Neuanlage von Ressourcen (siehe Abschnitt 1.8.2).

1.7.2 REST-Actions verwenden respond_to

Wir wissen jetzt, dass REST-Actions durch die Kombination von Ressource-URL und HTTP-Verb aktiviert werden. Das Resultat sind sauberere URLs, die nur die zu manipulierende Ressource adressieren, nicht aber die durchzuführende Aktion. Doch was zeichnet eine REST-Action neben der Art ihres Aufrufs noch aus?

Eine REST-Action zeichnet sich dadurch aus, dass sie auf unterschiedliche Client-Typen reagieren kann und unterschiedliche Response-Formate liefert. Typische Client-Typen einer Web-Anwendung sind neben Browser-Clients beispielsweise Web Service-Clients, die die Serverantwort im XML-Format erwarten oder auch RSS-Reader, die die Antwort gerne im RSS- oder Atom-Format hätten.

Zentrale Steuerinstanz für die Erzeugung des vom Client gewünschten Antwortformats ist die Methode *respond_to*, die der Scaffold-Generator bereits in die erzeugten CRUD-Actions hinein generiert hat. Dies verdeutlicht z.B. die *show*-Action:

Listing 1.2: ontrack/app/controllers/projects_controller.rb

```
# GET /projects/1
# GET /projects/1.xml
def show
  @project = Project.find(params[:id])
```

[4] Wir bezeichnen die verwendete HTTP-Methode im folgenden auch als *HTTP-Verb*, da hierdurch ein *Tun* im Sinne einer durchzuführenden Aktion ausgedrückt wird.

```
    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @project.to_xml }
    end
  end
```

Der Methode *respond_to* wird ein Block mit formatspezifischen Anweisungen übergeben. Im Beispiel behandelt der Block zwei Formate: HTML und XML. Je nachdem, welches Format der Client wünscht, werden die jeweiligen formatspezifischen Anweisungen ausgeführt. Im Fall von HTML wird gar nichts getan, d.h. es wird der Default-View *show.rhtml* ausgeliefert. Im Fall von XML wird die angeforderte Ressource nach XML konvertiert und in diesem Format an den Client geliefert.

Die Steuerung von *respond_to* arbeitet in zwei Varianten: Entweder wertet *respond_to* das Accept-Feld im HTTP-Header aus, oder der Aufruf-URL wird das gewünschte Format explizit mitgegeben.

1.7.3 Accept-Feld im HTTP-Header

Beginnen wir mir Variante 1, d.h. der Angabe des HTTP-Verbs im Accept-Feld des HTTP-Headers. Am einfachsten geht dies mit *curl*, einem HTTP-Kommandozeilenclient, mit dem sich unter anderem der HTTP-Header explizit setzen lässt. Bevor der Test mit *curl* beginnen kann, müssen Sie den Webserver starten:

```
> ruby script/server webrick
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-12-30 18:10:50] INFO  WEBrick 1.3.1
[2006-12-30 18:10:50] INFO  ruby 1.8.4 (2005-12-24) [i686-darwin8.6.1]
[2006-12-30 18:10:50] INFO  WEBrick::HTTPServer#start: pid=4709 port=3000
```

Erfassen Sie anschliessend ein oder mehrere Projekte über einen Browser (siehe Abbildung 1.2).

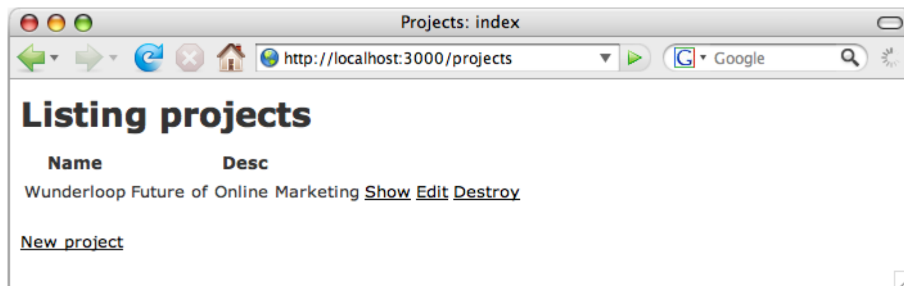

Abbildung 1.2: Initiale Projekterfassung

Der folgende curl-Aufruf fordert die Projekt-Ressource 1 im XML-Format an:

```
> curl -H "Accept: application/xml" \
```

```
    -i -X GET http://localhost:3000/projects/1
=>
HTTP/1.1 200 OK
Connection: close
Date: Sat, 30 Dec 2006 17:31:50 GMT
Set-Cookie: _session_id=4545eabd9d1bebde367ecbadf015bcc2; path=/
Status: 200 OK
Cache-Control: no-cache
Server: Mongrel 0.3.13.4
Content-Type: application/xml; charset=utf-8
Content-Length: 160

<?xml version="1.0" encoding="UTF-8"?>
<project>
  <desc>Future of Online Marketing</desc>
  <id type="integer">1</id>
  <name>Wunderloop</name>
</project>
```

Der Request wird vom Rails-Dispatcher auf die *show*-Action geroutet. Aufgrund des XML-Wunsches im Accept-Feld führt *respond_to* den format.xml-Block aus, der das angeforderte Projekt nach XML konvertiert und als Response liefert.

Curl eignet sich nicht nur zum Testen unterschiedlicher Response-Formate, sondern auch zum Senden von HTTP-Methoden, die vom Browser nicht unterstützt werden. Folgender Aufruf löscht z.B. die Project-Ressource mit der Id 1:

```
> curl -X DELETE http://localhost:3000/projects/1
=>
<html><body>You are being
<a href="http://localhost:3000/projects">redirected</a>.
</body></html>
```

Als zu verwendende HTTP-Methode wird dem Request explizit DELETE vorgegeben. Der Rails-Dispatcher wertet das HTTP-Verb aus und routet den Request auf die *destroy*-Action des *ProjectsController*s. Beachten Sie, dass die URL identisch zum vorherigen curl-Aufruf ist. Der einzige Unterschied ist das verwendete HTTP-Verb.

1.7.4 Formatangabe via URL

Die zweite Möglichkeit zur Anforderung unterschiedlicher Response-Formate ist die Erweiterung der URL um das gewünschte Format. Wenn Sie Projekt 1 nicht bereits durch den vorangegangenen Destroy-Aufruf gelöscht haben, können Sie die XML-Repräsentation des Projekts direkt im Browser anfordern:

```
http://localhost:3000/projects/1.xml
```

Achtung Mac-Benutzer: Der Aufruf lässt sich besser im Firefox als im Safari testen, da Safari die vom Server gelieferten XML-Tags ignoriert, während Firefox wunderschön formatiertes XML ausgibt (siehe Abbildung 1.3).

Abbildung 1.3: Projekt Wunderloop im XML-Format

Wir wissen jetzt, wie REST-Controller arbeiten und wie die zugehörigen Aufruf-URLs aussehen. In den folgenden beiden Abschnitten geht es um die Verwendung bzw. die Erzeugung dieser neuen URLs in Views und Controllern.

1.8 REST-URLs in Views

Views repräsentieren die Schnittstelle zwischen Anwendung und Benutzer. Der Benutzer interagiert über Links und Buttons mit der Anwendung. Traditionell werden Links mit Hilfe des Helpers *link_to* erzeugt. Die Methode erwartet eine Hash mit Controller, Action sowie einer Reihe optionaler Request-Parameter:

```
link_to :controller => "projects", :action => "show", :id => project
=>
<a href="/projects/show/1">Show</a>
```

Was unmittelbar auffällt ist, dass die traditionelle Verwendung von *link_to* nicht besonders gut mit dem REST-Ansatz korrespondiert, da REST Action-lose URLs bevorzugt. REST-URLs kennen keine Actions und daher kommt es darauf an, dass die von Links und Buttons ausgelösten Requests mit der – im Sinne von REST – richtigen HTTP-Methode an den Server übertragen werden.

Rails löst diese Anforderung, indem Links zwar weiterhin mit *link_to* erzeugt werden, die ehemals übergebene Hash aber durch den Aufruf einer *Path*-Methode ersetzt wird. Path-Methoden erzeugen Verweisziele, die von *link_to* im href-Attribut des erzeugten Links eingesetzt werden. Als erstes Beispiel dient ein Link auf die *show*-Action im *ProjectsController*. Statt Controller, Action und Projekt-Id wird *project_path(:id)* verwendet:

```
link_to "Show", project_path(project)
=>
<a href="/projects/1">Show</a>
```

Während das href-Attribut eines traditionellen *link_to*-Aufrufs noch Controller und Action enthielt, enthält das von *project_path* erzeugte HTML-Element nunmehr nur noch den Controller nebst der referenzierten Projekt-Id und damit eine typische REST-URL. Der Rails-Dispatcher erkennt aufgrund des für Links standardmäßig

verwendeten GET-Aufrufs, dass das gewählte Projekt angezeigt, d.h. die *show*-Action ausgeführt werden soll.

Für jede Ressource erzeugt Rails sieben Standard Path-Methoden, die in nachfolgender Tabelle zusammengefasst sind:

Tabelle 1.2: Standart Path-Methoden

Path-Methode	HTTP-Verb	Pfad	Aufzurufende Action
projects_path	GET	/projects	index
project_path(1)	GET	/projects/1	show
new_project_path	GET	/projects/new	new
edit_project_path(1)	GET	/projects/1;edit	edit
projects_path	POST	/projects	create
project_path(1)	PUT	/projects/1	update
project_path(1)	DELETE	/projects/1	destroy

Jede Path-Methode ist mit einem HTTP-Verb assoziiert, d.h. der HTTP-Methode, mit der ein Request beim Klick auf den entsprechenden Link bzw. Button an den Server gesendet wird. Einige der Requests (show, update) werden bereits per Default mit dem richtigen HTTP-Verb übertragen (hier GET und POST). Andere (create, destroy) bedürfen einer Sonderbehandlung (Hidden Fields), da der Browser PUT und DELETE bekanntlich gar nicht kennt. Mehr über diese Sonderbehandlung und deren Implementierung erfahren Sie in den Abschnitten 1.8.3 und 1.8.2.

Ein weiterer Blick in die Tabelle zeigt ausserdem, dass die vier HTTP-Verben nicht genügen, um sämtliche CRUD-Actions abzubilden. Während die ersten beiden Methoden noch gut mit GET funktionieren und auf jeweils eindeutige Actions geroutet werden, sieht es bei *new_project_path* und *edit_project_path* schon anders aus.

1.8.1 New und Edit

Ein Klick auf den New-Link wird via GET an der Server übertragen. Das folgende Beispiel zeigt, dass der erzeugte Pfad neben aufzurufendem Controller auch die zu aktivierende Action *new* enthält:

```
link_to "New", new_project_path
=>
<a href="/projects/new">New</a>
```

Ist dies etwa ein Bruch im REST-Ansatz? Auf den ersten Blick vielleicht. Bei näherer Betrachtung wird jedoch klar, dass *new* im engeren Sinne keine REST/CRUD-Action ist, sondern eher eine Art Vorbereitungs-Action zum Erzeugen einer neuen Ressource. Die eigentliche CRUD-Action *create* wird erst beim Submit des New-Formulars durchgeführt. Ausserdem fehlt in der URL die Ressource-Id, da es die Ressource ja noch gar nicht gibt. Eine URL ohne Id ist keine REST-URL, da REST-URLs Ressourcen eindeutig adressieren. Die Action dient also ausschliesslich der Anzeige eines Neuanlage-Formulars.

Gleiches gilt für die Methode *edit_project_path*: Diese bezieht sich zwar auf eine konkrete Ressourcen-Instanz, dient aber auch nur zur Vorbereitung der eigentlichen CRUD-Action *update*. Das heisst auch hier wird die eigentliche CRUD-Action erst beim Submit des Edit-Formulars durchgeführt. Anders als *new_project_path* benötigt *edit_project_path* aber die Id der zu bearbeitenden Ressource. Gemäss der REST-Konvention folgt diese dem Controller: */projects/1*. Einfach per GET abgesetzt, würde dieser Aufruf auf die show-Action geroutet. Um dies zu vermeiden, fügt *edit_project_path* dem erzeugten Pfad die aufzurufende Action an. Der erzeugte HTML-Link sieht entsprechend folgendermassen aus:

```
link_to "Edit", edit_project_path(project)
=>
<a href="/projects/1;edit">Edit</a>
```

Es ist also sowohl für *new* als auch für *edit* in Ordnung, dass die Action in der URL auftaucht, da es sich bei beiden im engeren Sinne um keine REST/CRUD-Actions handelt. Das gleiche Prinzip wird übrigens für eigene, d.h. von den Standard CRUD-Namen abweichende Actions verwendet. Dazu mehr in Abschnitt 1.12.

1.8.2 Path-Methoden in Formularen: Create und Update

Traditionell werden Formulare in Rails mit dem Helper *form_tag* bzw. *form_for* unter Angabe einer Submit-Action erstellt:

```
<% form_for :project, @project, :url => { :action => "create" } do |f| %>
...
<% end %>
```

In REST-Anwendungen wird die :url-Hash den durch Aufruf einer Path-Methode ersetzt:

- *project_path* für Neuanlage-Formulare und
- *project_path(:id)* für Editier-Formulare

Neuanlage-Formulare

Formulare werden standardmäßig per POST an den Server übertragen. Der Aufruf von *project_path* ohne Id resultiert somit in den Pfad /projects, der per POST übertragen zum Aufruf der Action *create* führt:

```
form_for(:project, :url => projects_path) do |f| ...
=>
<form action="/projects" method="post">
```

Editier-Formulare

Gemäß den REST-Vorgaben werden Updates per PUT übertragen. Allerdings wissen wir auch, dass PUT genau wie DELETE vom Browser nicht unterstütz wird. Die Rails-Lösung ist auch hier die Verwendung des Keys *method* in der *html*-Hash von *form_for*:

```
form_for(:project, :url => project_path(@project),
         :html => { :method => :put }) do |f| ...
=>
<form action="/projects/1" method="post">
<div style="margin:0;padding:0">
  <input name="_method" type="hidden" value="put" />
</div>
```

Rails generiert ein Hidden-Field *_method* mit der zu verwendenden HTTP-Methode *put*. Der Dispatcher wertet das Feld aus und aktiviert die zugehörige Action *update*.

1.8.3 Destroy

Beachten Sie, dass die zum Anzeigen und Löschen von Projekten verwendete Methode in beiden Fällen *project_path* ist:

```
link_to "Show", project_path(project)
link_to "Destroy", project_path(project), :method => :delete
```

Allerdings übergibt der Destroy-Link zusätzlich den Parameter *:method* zur Angabe der zu verwendenden HTTP-Methode (hier :delete). Da der Browser DELETE nicht unterstützt, generiert Rails ein JavaScript-Fragment, das beim Klick auf den Link ausgeführt wird:

```
link_to "Destroy", project_path(project), :method => :delete
=>
<a href="/projects/1"
  onclick="var f = document.createElement('form');
  f.style.display = 'none'; this.parentNode.appendChild(f);
  f.method = 'POST'; f.action = this.href;
  var m = document.createElement('input');
  m.setAttribute('type', 'hidden');
  m.setAttribute('name', '_method');
  m.setAttribute('value', 'delete'); f.appendChild(m);f.submit();
  return false;">Destroy</a>
```

Das Script erzeugt ein Formular und sorgt dafür, dass das gewünschte HTTP-Verb *delete* im Hidden-Field *_method* an den Server übertragen wird. Der Rails-Dispatcher wertet den Inhalt dieses Felds aus und erkennt so, dass der Request auf die Action *destroy* geroutet werden muss.

1.9 URL-Methoden im Controller

Genau wie in REST-Views Links und Submit-Actions mit Hilfe neuer Helper erzeugt werden, müssen Controller die neue Verweistechnik beim Durchführen von Redirects beachten. Diesem Zweck dienen die so genannten URL-Methoden, von denen Rails zu jeder Path-Methode ein entsprechendes Gegenstück generiert, z.B.

`project_url` für `project_path`

oder

`projects_url` für `projects_path`.

Im Gegensatz zu Path-Methoden erzeugen URL-Methoden vollständige URLs inklusive Protokoll, Host, Port und Pfad:

```
project_url(1)
=>
"http://localhost:3000/projects/1"

projects_url
=>
"http://localhost:3000/projects"
```

URL-Methoden werden in den Controllern einer REST-Anwendungen überall dort verwendet, wo der *redirect_to*-Methode traditionell eine Controller-Action-Parameter Hash übergeben wurde. Aus

```
redirect_to :action => "show", :id => @project.id
```

wird in einer REST-Anwendung:

```
redirect_to project_url(@project)
```

Ein Beispiel hierfür finden Sie in der *destroy*-Action, in der *projects_url* ohne Parameter verwendet wird, um nach dem Löschen eines Projekts auf die Liste aller Projekte zu redirecten:

Listing 1.3: ontrack/app/controllers/projects_controller.rb

```
def destroy
  @project = Project.find(params[:id])
  @project.destroy

  respond_to do |format|
    format.html { redirect_to projects_url }
    format.xml  { head :ok }
  end
end
```

1.10 REST-Routing

Jetzt haben wir Ihnen neben den REST-Konzepten eine Reihe neuer Methoden zur Verwendung in Links, Formularen und Controllern erklärt, bisher aber völlig offen gelassen, woher diese Methoden eigentlich kommen. Verantwortlich dafür, dass die beschriebenen Methoden existieren und ihr Aufruf zu korrekt gerouteten Verweiszielen führen, ist ein neuer Eintrag in der Routing-Datei *config/routes.rb*:

```
map.resources :projects
```

Der Eintrag wurde vom Scaffold-Generator in die Datei eingetragen und erzeugt genau die benannten Routen, die zum Aufruf der REST-Actions des *ProjectsControllers* vorhanden sein müssen.

Ausserdem erzeugt *resources* die Path- und URL-Methoden für die Ressource *Project*, die Sie im vorherigen Abschnitt kennen gelernt haben:

```
map.resources :projects
=>
Route         Erzeugte Helper
-----------------------------------------------------------
projects      projects_url, projects_path
project       project_url(id), project_path(id)
new_project   new_project_url, new_project_path
edit_project  edit_project_url(id), edit_project_path(id)
```

1.10.1 Konventionen

Eine notwendige Konsequenz aus der Verwendung von REST-Routen ist die Einhaltung von Namenskonventionen bei der Benennung von CRUD-Actions. Der folgende *link_to*-Aufruf und das daraus generierte HTML verdeutlichen dies:

```
link_to "Show", project_path(project)
=>
<a href="/projects/1">Show</a>
```

Weder der *link_to*-Aufruf noch der erzeugte HTML-Code enthalten den Namen der aufzurufenden Controller-Action. Der Rails-Dispatcher weiss aber, dass die Route */projects/:id* auf die *show*-Action des *ProjectsController*s zu routen ist, wenn der zugehörige Request via GET gesendet wurde. Demzufolge muss der Controller über eine Action mit dem Namen *show* verfügen. Die selbe Konvention gilt auch für die Actions *index*, *update*, *create*, *destroy*, *new* und *edit*, d.h. REST-Controller müssen diese Methoden implementieren.

1.10.2 Customizing

REST-Routen können mit Hilfe folgender Optionen an anwendungsspezifische Anforderungen angepasst werden:

- **:controller.** Gibt den zu verwendenden Controller an.
- **:path_prefix.** Gibt das URL-Präfix inklusive der erforderlichen Variablen an.
- **:name_prefix.** Gibt das Präfix der erzeugten Routen-Helper an.
- **:singular** Gibt den Singular-Namen an, der für die Member Route verwendet werden soll.

Der folgende Routing-Eintrag erzeugt Routen für die neue Ressource *Sprint*. Sprint steht hier als Synonym für Iteration und mapped auf das selbe ActiveRecord-Modell *Iteration*, dass wir im folgenden Abschnitt einführen:

```
map.resources :sprints,
              :controller => "ontrack",
              :path_prefix => "/ontrack/:project_id",
              :name_prefix => "ontrack_"
```

Die Option *path_prefix* gibt das URL-Format vor. Jede URL-beginnt mit */ontrack* gefolgt von einer Projekt-Id. Als verantwortlicher Controller wird *OntrackController* festgelegt. Die URL

```
http://localhost:3000/ontrack/1/sprints
```

wird entsprechend den Routing-Vorgaben auf die *index*-Action des *OntrackControllers* geroutet. Ein anderes Beispiel ist die URL

```
http://localhost:3000/ontrack/1/sprints/1,
```

die auf die *show*-Action des *OntrackControllers* geroutet wird.

Während path_prefix das Format der URL festlegt, sorgt name_prefix dafür, dass sämtliche generierten Helper-Methoden mit *ontrack_* beginnen, z.B.:

```
ontrack_sprints_path(1)
=>
/ontrack/1/sprints
```

oder

```
ontrack_edit_sprint_path(1, 1)
=>
/ontrack/1/sprints/1;edit
```

1.11 Verschachtelte Ressourcen

Richtig interessant wird REST-basierte Entwicklung bei der Verwendung von verschachtelten Ressourcen[5]. Hierbei wird zum einen die Bedeutung von sauberen URLs noch einmal richtig deutlich, und zum anderen helfen verschachtelte Ressourcen dabei, REST und die Bedeutung dieses Paradigmas besser zu verstehen.

Verschachtelte Ressourcen sind eng gekoppelte Ressourcen im Sinne eine Eltern-Kind-Beziehung. Im Kontext von Rails sind dies Modelle, die z.B. in einer 1:N-Beziehung zu einander stehen, wie beispielsweise Projekte und Iterationen in Ontrack. Controller zeichnen sich weiterhin für die Manipulation eines einzigen Modelltyps verantwortlich, beziehen dabei aber im Falle des Kind-Controllers das Modell der Eltern-Ressource lesend mit ein. Klingt kompliziert, wird aber im Verlauf dieses Abschnitts verständlich werden.

Der REST-Ansatz in Rails reflektiert die Relationen zwischen verschachtelten Ressourcen in den URLs und bewahrt dabei den *sauberen* Charakter von REST-URLs. Lassen Sie uns dieses Prinzip am Beispiel von Iterationen und Projekten in Ontrack

[5] In der englischsprachigen Dokumentation werden diese als *nested resources* bezeichnet.

beschreiben. Los geht es mit der Generierung der neuen Ressource *Iteration* sowie der Erzeugung der zugehörigen Datenbanktabelle *iterations*:

```
> ruby script/generate scaffold_resource iteration name:string \
       start:date end:date project_id:integer
> rake db:migrate
```

Projekte stehen in einer 1:N-Beziehung zu Iterationen. Diese Beziehung wird auf Modellebene implementiert:

Listing 1.4: ontrack/app/models/project.rb

```
class Project < ActiveRecord::Base
  has_many :iterations
end
```

Listing 1.5: ontrack/app/models/iteration.rb

```
class Iteration < ActiveRecord::Base
  belongs_to :project
end
```

Der Generator erzeugt neben den bekannten Artefakten wie Modell, Controller und Views einen weiteren Routen-Eintrag in *config/routes.rb*:

```
map.resources :iterations
```

Der Eintrag erzeugt analog zu Projekten neue Routen und Helper für die Manipulation der Ressource *Iteration*. Allerdings sind Iterationen nur im Kontext eines zuvor ausgewählten Projekts sinnvoll. Diese Tatsache wird von den erzeugten Routen und Helpern nicht berücksichtigt. Beispielsweise erzeugt der Helper *new_iteration_path* den Pfad */iterations/new*, der keinerlei Informationen darüber enthält, für welches Projekt die neue Iteration erzeugt werden soll.

Der Clou verschachtelter Ressourcen ist aber insbesondere die Erkenntnis, dass die Kind-Ressource (hier Iteration) ohne die zugehörige Eltern-Ressource (hier Project) keinen Sinn ergibt. Und genau diesen Sachverhalt versucht REST-Rails in den verwendeten URLs sowie im Controller der Kind-Ressource zu reflektieren. Damit dies funktioniert, muss der generierte resources-Eintrag in *config/routes.rb* ersetzt werden:

```
map.resources :iterations
```

wird ersetzt durch:

```
map.resources :projects do |projects|
  projects.resources :iterations
end
```

Der Eintrag macht *Iteration* zu einer verschachtelten Ressource und erzeugt eine Reihe neuer Routen, die den Zugriff auf Iterationen immer im Kontext eines Projekts erfolgen lassen. Die erzeugten Routen haben folgendes Format:

```
/project/:project_id/iterations
/project/:project_id/iterations/:id
```

Beispielsweise führt die Eingabe der URL

```
http://localhost:3000/projects/1/iterations
```

zum Aufruf der *index*-Action des *IterationsController*s, der die Id des Projektes im Request-Parameter *:project_id* erhält. Beachten Sie den Charakter der URL, der sehr stark an die zugrunde liegende ActiveRecord-Assoziation erinnert und diese widerspiegelt:

```
/projects/1/iterations <=> Project.find(1).iterations
```

Verschachtelte REST-URLs sind weiterhin reine REST-URLs, d.h. sie adressieren Ressourcen und keine Aktionen. Die Tatsache, dass es sich um eine verschachtelte Ressource handelt, wird dadurch ausgedrückt, dass zwei REST-URLs aneinander gehängt werden, wie z.B. der Aufruf der *show*-Action verdeutlicht:

```
http://localhost:3000/projects/1/iterations/1
```

1.11.1 Anpassung des Controllers

Der generierte *IterationsController* weiss nichts davon, dass er jetzt ein Controller für eine verschachtelte Ressource ist und in jedem Request die Id des zugehörigen Projekts mitgeteilt bekommt. Beispielsweise lädt die *index*-Action immer noch alle gespeicherten Iterationen, obwohl die Aufruf-URL zum Ausdruck bringt, dass nur die Iterationen des angegebenen Projekts geladen werden sollen:

Listing 1.6: ontrack/app/controllers/iterations_controller.rb

```
def index
  @iterations = Iteration.find(:all)

  respond_to do |format|
    format.html # index.rhtml
    format.xml  { render :xml => @iterations.to_xml }
  end
end
```

Die Action muss entsprechend umgebaut werden, so dass nur noch die Iterationen des gewählten Projekts geladen werden:

Listing 1.7: ontrack/app/controllers/iterations_controller.rb

```
def index
  project = Project.find(params[:project_id])
  @iterations = project.iterations.find(:all)
  ...
end
```

Sämtliche *IterationsController*-Actions funktionieren jetzt nur noch mit dem Präfix */projects/:project_id*. Es wird somit immer der Projektkontext festgelegt, auf den sich die Manipulation der Ressource *Iteration* bezieht. Das bedeutet aber auch, dass neben *index* auch die Actions *create* (siehe Abschnitt 1.11.3) und *update* angepasst (siehe Abschnitt 1.11.4) werden müssen.

1.11.2 Neue Parameter für Path- und URL-Helper

Neben den Routen erzeugt der neue resource-Eintrag in *config/routes.rb* neue Helper, die genau wie die Route, die Projekt-Id als ersten Parameter erwarten. Zum Beispiel wird die Liste aller Iterationen eines Projekts vom Helper *iterations_path* erzeugt. Die Namen der Helper bleiben identisch zu denen von nicht-verschachtelten Ressourcen. Was sich jedoch ändert, ist die Anzahl der zu übergebenden Parameter. Helper für verschachtelte Ressourcen erwarten als ersten Parameter immer die Id der umschliessenden Ressource, in unserem Beispiel also die Projekt-Id. Dies zeigt z.B. der *Iterations*-Link, der, eingebaut in den *index*-View des *ProjectsController*s, alle Iterationen des gewählten Projekts anzeigt. Hier wird die URL auf die Iterationenliste vom Helper *iterations_path* erzeugt, der als Parameter die Projekt-Id erwartet:

```
link_to "Iterations", iterations_path(project)
=>
<a href="/projects/1/iterations">Iterations</a>
```

Zum besseren Verständnis hier noch einmal der Link, eingebaut in den *index*-View des *ProjectsController*s:

Listing 1.8: ontrack/app/views/projects/index.rhtml

```
...
<% for project in @projects %>
  <tr>
    <td><%=h project.name %></td>
    <td><%=h project.desc %></td>
    <td><%= link_to "Iterations", iterations_path(project) %></td>
    <td><%= link_to "Show", project_path(project) %></td>
    <td><%= link_to "Edit", edit_project_path(project) %></td>
    <td><%= link_to "Destroy", project_path(project),
            :confirm => "Are you sure?", :method => :delete %></td>
  </tr>
<% end %>
...
```

Die Folge der geänderten Parameterreihenfolge für Iterations-Helper ist, dass neben einigen Actions im Controller, auch viele der Scaffold-Views für Iterationen nicht mehr funktionieren. Zum Beispiel enthält der index-View eine Tabelle mit sämtlichen Iterationen und drei Links für jede Iteration:

Listing 1.9: ontrack/app/views/iterations/index.rhtml

```
...
<% for iteration in @iterations %>
  <tr>
    <td><%=h iteration.name %></td>
    <td><%=h iteration.start %></td>
    <td><%=h iteration.end %></td>
    <td><%= link_to "Show", iteration_path(iteration) %></td>
    <td><%= link_to "Edit", edit_iteration_path(iteration) %></td>
    <td><%= link_to "Destroy", iteration_path(iteration),
            :confirm => "Are you sure?", :method => :delete %></td>
  </tr>
<% end %>
...
```

Allen Links wird als erster und einziger Parameter die Id der jeweiligen Iteration übergeben, was nicht mehr funktionieren kann, da der erste Parameter eines Iteration-Helper immer die Projekt-Id ist. Die notwendige Änderung sieht wie folgt aus:

Listing 1.10: ontrack/app/views/projects/index.rhtml

```
...
<% for iteration in @iterations %>
  <tr>
    <td><%=h iteration.name %></td>
    <td><%=h iteration.start %></td>
    <td><%=h iteration.end %></td>
    <td><%= link_to "Show", iteration_path(iteration.project,
              iteration) %></td>
    <td><%= link_to "Edit", edit_iteration_path(iteration.project,
              iteration) %></td>
    <td><%= link_to "Destroy", iteration_path(iteration.project,
              iteration), :confirm => "Are you sure?",
              :method => :delete %></td>
  </tr>
<% end %>
...
```

Alternativ zur Einhaltung der Parameter-Reihenfolge kann den Helpern verschachtelter Ressourcen eine Hash mit Id-Werten übergeben werden:

```
iteration_path(:project_id => iteration.project, :id => iteration)
```

Diese Schreibweise erhöht die Lesbarkeit des Codes, wenn nicht unmittelbar klar ist, zu welchem Typ von Objekt Iterationen in Beziehung stehen.

1.11.3 Zufügen neuer Iterationen

Das Zufügen neuer Iterationen funktioniert nur noch im Kontext eines zuvor gewählten Projekts. Damit dies einfach möglich ist, wird der *index*-View des *ProjectsController*s für jedes angezeigte Projekt um einen *New Iteration*-Link erweitert:

Listing 1.11: ontrack/app/views/projects/index.rhtml

```
...
<% for project in @projects %>
  <tr>
    <td><%=h project.name %></td>
    <td><%=h project.desc %></td>
    <td><%= link_to "Iterations", iterations_path(project) %></td>
    <td><%= link_to "Show", project_path(project) %></td>
    <td><%= link_to "Edit", edit_project_path(project) %></td>
    <td><%= link_to "Destroy", project_path(project),
            :confirm => "Are you sure?", :method => :delete %></td>
    <td><%= link_to "New Iteration", new_iteration_path(project) %></td>
  </tr>
<% end %>
...
```

Als Path-Methode wird *new_iteration_path* verwendet, die für das Projekt mit der Id 1 folgendes HTML generiert:

```
link_to "New Iteration", new_iteration_path(project)
=>
<a href="/projects/1/iterations/new">New iteration</a>
```

Der Link wird auf die *new*-Action des *IterationsController*s geroutet. Die Action erhält als Request-Parameter *project_id* den Wert 1, d.h. die Id des Projekts, für das der Link geklickt wurde. Die Projekt-Id steht damit im gerenderten View *new* des *IterationsController*s zur Verfügung und kann dort im Helper *iterations_path* verwendet werden, der für die Erzeugung der Form-Action zuständig ist. Das vom Helper erzeugte Formular enthält im Attribut *action* eine verschachtelte Route mit der Id des Projekts, für das die neue Iteration erzeugt werden soll:

Listing 1.12: ontrack/app/views/iterations/new.rhtml

```
<% form_for(:iteration,
            :url => iterations_path(params[:project_id])) do |f| %>
...
<% end %>
=>
<form action="/projects/1/iterations" method="post">
```

Die Übergabe von *params[:project_id]* in *iterations_path* ist optional, da Rails automatisch den Request-Parameter *project_id* in das generierte action-Attribut einsetzen würde, d.h.

```
form_for(:iteration, :url => iterations_path)
```

hat den gleichen Effekt.

Das REST-Routing sorgt dafür, dass die Form-Action */projects/1/iterations* in Kombination mit der HTTP-Methode POST zum Aufruf der *create*-Action im *IterationsController* führt. Die im Form-Tag angegebene HTTP-Methode (method='post') wird vom Helper standardmäßig erzeugt, da kein expliziter Wert angegeben wurde und *post* der Default-Wert ist.

Der create-Action wird neben den eigentlichen Form-Parametern die Projekt-Id im Request-Parameter *project_id* übergeben. Entsprechend muss die Methode angepasst werden, damit die Iteration dem gewählten Projekt zugeordnet wird:

Listing 1.13: ontrack/app/controllers/iterations_controller.rb

```
1  def create
2    @iteration = Iteration.new(params[:iteration])
3    @iteration.project = Project.find(params[:project_id])
4
5    respond_to do |format|
6      if @iteration.save
7        flash[:notice] = "Iteration was successfully created."
8        format.html { redirect_to iteration_url(@iteration.project,
9                      @iteration) }
10       format.xml  { head :created, :location =>
11                     iteration_url(@iteration.project, @iteration) }
12     else
13       format.html { render :action => "new" }
14       format.xml  { render :xml => @iteration.errors.to_xml }
15     end
16   end
17 end
```

In Zeile 3 wird das Projekt explizit zugewiesen. Ausserdem wurden in Zeile 8 und 11 die Helper *iteration_url* um die Angabe der Projekt-Id erweitert.

Damit das Zufügen neuer Iterationen jetzt auch wirklich funktioniert, müssen Sie die Links *Edit* und *Back* im *Show*-View des *IterationsController*s noch um den Projekt-Id Parameter erweitern:

Listing 1.14: ontrack/app/views/iterations/show.rhtml

```
...
<%= link_to "Edit", edit_iteration_path(@iteration.project, @iteration) %>
<%= link_to "Back", iterations_path(@iteration.project) %>
```

Dieser View wird nämlich nach der Neuanlage gerendert und produziert einen Fehler, wenn die Projekt-Id nicht übergeben wird.

1.11.4 Bearbeiten existierender Iterationen

Zum Bearbeiten von Iterationen sind zwei Anpassungen in den generierten Sourcen notwendig.

Dem *form_for*-Helper im *edit*-View des *IterationsController*s wird nur die Iterations-Id anstelle der erforderlichen Projekt- und Iterations-Id übergeben:

```
form_for(:iteration,
         :url => iteration_path(@iteration),
         :html => { :method => :put }) do |f|
```

Der Aufruf muss wie folgt geändert werden:

```
form_for(:iteration,
         :url => iteration_path(params[:project_id], @iteration),
         :html => { :method => :put }) do |f|
```

Ein ähnliche Änderung ist in der vom Formular aufgerufenen *update*-Action durchzuführen. Auch hier wird der Methode *iteration_url* in Zeile 7 nach erfolgreichem Update nur die Iterations-Id übergeben:

Listing 1.15: ontrack/app/controllers/iterations_controller.rb

```
1  def update
2    @iteration = Iteration.find(params[:id])
3
4    respond_to do |format|
5      if @iteration.update_attributes(params[:iteration])
6        flash[:notice] = "Iteration was successfully updated."
7        format.html { redirect_to iteration_url(@iteration) }
8        format.xml  { head :ok }
9      else
10       format.html { render :action => "edit" }
11       format.xml  { render :xml => @iteration.errors.to_xml }
12     end
13   end
14 end
```

Analog zum View wird Zeile 7 geändert in:

```
format.html { redirect_to iteration_url(@iteration.project,
                                        @iteration) }
```

Die Create- und Update-Views und -Actions sind nach diesen Änderungen soweit wieder lauffähig, dass Iterationen angelegt und bearbeitet werden können. Durchforsten Sie sicherheitshalber noch einmal den *IterationsController* und dessen Views nach Path- und URL-Helpern, denen noch keine Projekt-Id übergeben wird, und ändern Sie diese entsprechend ab.

1.12 Eigene Actions

Der *resources*-Eintrag in der Routing-Datei erzeugt benannte Routen und Helper für CRUD-Actions. Doch wie sieht es mit Routen und Helpern für Actions aus, die keine CRUD-Actions sind und schliesslich auch in die Controller gehören? Als Beispiel dient eine *close*-Action im *ProjectsController*, die ein Projekt schliesst, d.h. als abgeschlossen kennzeichnet. Zunächst die Datenbankmigration:

```
> ruby script/generate migration add_closed_to_projects
    exists db/migrate
    create db/migrate/003_add_closed_to_projects.rb
```

Listing 1.16: ontrack/db/migrate/003_add_closed_to_projects.rb

```
class AddClosedToProjects < ActiveRecord::Migration
  def self.up
    add_column :projects, :closed, :boolean, :default => false
  end

  def self.down
    remove_column :projects, :closed
  end
end

rake db:migrate
```

Als nächstes wird ein Close-Link im *index*-View des *ProjectsController*s benötigt:

Listing 1.17: ontrack/app/views/projects/index.rhtml

```
<% for project in @projects %>
  <tr id="project_<%= project.id %>">
    <td><%=h project.name %></td>
    <td><%= link_to "Show", project_path(project) %></td>
    ...
    <td><%= link_to "Close", <WELCHER_HELPER?> %></td>
  </tr>
<% end %>
```

Mit Einfügen des Links stellen sich zwei Fragen:

1. Mit welchem HTTP-Verb soll *close* gesendet werden?
2. Woher kommt der Helper zum Erzeugen des Pfads auf die *close*-Action?

Da es sich bei *close* nicht um eine typische CRUD-Action handelt, weiss Rails nicht, welches HTTP-Verb verwendet werden soll. Close aktualisiert das Projekt, d.h. ist eine Art Update und sollte im Sinne von REST via POST gesendet werden. Die Route und zugehörigen Helper definieren wir in der Routing-Datei *config/routes.rb* mit Hilfe der *member*-Hash im *resources*-Aufruf für *projects*. Die Hash besteht aus einer Menge von Action-Methoden-Paaren und spezifiziert, welche Action mit welchem HTTP-Verb aufgerufen werden soll bzw. darf[6].

Als Werte sind *:get*, *:put*, *:post*, *:delete* und *:any* zulässig, wobei eine durch *:any* gekennzeichnete Action von jeder HTTP-Methode aufgerufen werden darf. In unserem Beispiel soll *close* via POST aufgerufen werden, so dass der *resources*-Eintrag wie folgt angepasst werden muss:

```
map.resources :projects, :member => { :close => :post }
```

Nach Anpassung des Eintrags steht unter anderem der neue Helper *close_project_path* zur Verfügung, der in den weiter oben beschriebenen Link eingebaut werden kann:

[6] Rails belegt die Routen mit HTTP-Restriktionen, so dass Aufrufe von Actions mit falschem HTTP-Verb zu RoutingError-Exceptions führen.

```
<td><%= link_to "Close", close_project_path(project) %></td>
```

Allerdings resultiert ein Klick auf den Link in einen Routing-Error:

```
no route found to match "/projects/1;close" with {:method=>:get}
```

Die Route ist zwar vorhanden, der neue *resources*-Eintrag sorgt aber dafür, dass ein entsprechender Request nur via HTTP-POST gesendet werden darf. Andere HTTP-Methoden, und so auch die vom neuen Link verwendete GET-Methode, werden von Rails abgelehnt. Was wir brauchen, ist ähnlich wie beim Destroy-Link, ein Helper, der ein Formular erzeugt, und dieses via POST an den Server sendet. Rails verfügt mit *button_to* über einen Helper, der genau dies tut:

```
<td><%= button_to "Close", close_project_path(project) %></td>
=>
<td>
  <form method="post" action="/projects/1;close" class="button-to">
    <div><input type="submit" value="Close" /></div>
  </form>
</td>
```

Jetzt fehlt nur noch die *close*-Action:

Listing 1.18: ontrack/app/controllers/projects_controller.rb

```
def close
  respond_to do |format|
    if Project.find(params[:id]).update_attribute(:closed, true)
      flash[:notice] = "Project was successfully closed."
      format.html { redirect_to projects_path }
      format.xml  { head :ok }
    else
      flash[:notice] = "Error while closing project."
      format.html { redirect_to projects_path }
      format.xml  { head 500 }
    end
  end
end
```

Neben *:member* können im *resources*-Aufruf die Keys *:collection* und *:new* verwendet werden. Die Verwendung von *:collection* ist dort erforderlich, wo sich die Action nicht auf eine einzelne, sondern auf eine Menge von Ressourcen bezieht, wie z.B. das Anfordern der Projektliste als RSS-Feed:

```
map.resources :projects, :collection => { :rss => :get }
--> GET /projects;rss (mapped auf die #rss action)
```

Schliesslich bleib der Hash-Key *:new*, zur Verwendung für Actions, die sich auf neue, d.h. noch nicht gespeicherte Ressourcen beziehen:

```
map.resources :projects, :new => { :validate => :post }
--> POST /projects/new;validate (mapped auf die #validate action)
```

1.12.1 Sind wir noch DRY?

Ein wenig klingt das hier beschriebene Procedere schon nach einer Verletzung des DRY-Prinzips: Schliesslich werden Actions nicht mehr nur im Controller implementiert, sondern zusätzlich in der Routing-Datei benannt.

Alternativ zum beschriebenen REST-konformen Vorgehen liessen sich Nicht-REST-Actions auch traditionell, d.h. per Angabe von Action und Projekt-Id aufrufen:

```
<%= link_to "Close", :action => "close", :id => project %>
```

Die hierfür benötigten Routen sind noch vorhanden, sofern Sie nicht den *map.connect ':controller/:action/:id'*-Aufruf aus der Routing-Datei gelöscht haben. Die alte Route funktioniert allerdings nur, wenn Sie nicht bereits den *resources*-Aufruf für *projects* wie oben beschrieben verändert haben.

1.13 Eigene Formate

Die Methode *respond_to* kennt standardmäßig folgende Formate:

```
respond_to do |wants|
  wants.text
  wants.html
  wants.js
  wants.ics
  wants.xml
  wants.rss
  wants.atom
  wants.yaml
end
```

Darüber hinaus können eigene Formate als MIME-Typen registriert werden. Angenommen, Sie haben eine Adressverwaltung programmiert und wollen die gespeicherten Adressen im vcard-Format[7] ausliefern. Dazu muss das neue Format als erstes in der Konfigurationsdatei *config/environment.rb* registriert werden:

```
Mime::Type.register "application/vcard", :vcard
```

Anschliessend kann die *show*-Action des *AddressesController*s so erweitert werden, dass Adressen im vcard-Format geliefert werden, wenn dies vom Client so gewünscht wird:

```
def show
  @address = Address.find(params[:id])

  respond_to do |format|
    format.vcard { render :xml => @address.to_vcard }
    ...
  end
end
```

[7] http://microformats.org/wiki/hcard

Die Methode *vcard* ist keine Standard ActiveRecord-Methode und muss gemäss der vcard-Spezifikation (RFC2426) implementiert werden. Wurde die Methode korrekt implementiert, dann sollte der Aufruf folgender URL eine Adresse im vcard-konformen XML-Format liefern:

```
http://localhost:3000/addresses/1.vcard
```

1.14 RESTful AJAX

In Bezug auf die Erstellung von REST-basierten AJAX-Anwendungen gibt es nicht viel neues zu lernen. Sie können die bekannten Remote-Helper verwenden und dem :url-Parameter statt Controller und Action-Hash eine Path-Methode übergeben. Der folgende Codeschnipsel macht aus dem *destroy*-Link im *index*-View des *ProjectsController*s einen AJAX-Link:

```
link_to_remote "Destroy", :url => project_path(project),
                          :method => :delete
=>
<a href="#" onclick="new Ajax.Request("/projects/1",
  {asynchronous:true, evalScripts:true, method:"delete"});
  return false;">Async Destroy</a>
```

Denken Sie bitte daran, die benötigten JavaScript-Bibliotheken einzubinden, wenn Sie nicht wie ich eine Viertelstunde rum probieren wollen, um rauszukriegen, warum der Link nicht funktioniert. Eine Variante ist ein Eintrag in der Layout-Datei des *ProjectsController*s *projects.rhtml*:

Listing 1.19: ontrack/app/views/layouts/projects.rhtml

```
<head>
  <%= javascript_include_tag :defaults %>
  ...
</head>
```

Ein Klick auf den Link wird auf die *destroy*-Action des *ProjectsController*s geroutet. Aus Sicht der Geschäftslogik macht die Methode bereits in ihrer ursprünglich generierten Form alles richtig, d.h. sie löscht das ausgewählte Projekt. Was noch fehlt, ist ein zusätzlicher Eintrag im *respond_to*-Block, der das vom Client gewünschte Format liefert, in diesem Fall JavaScript. Der folgende Codeschnipsel zeigt die bereits erweiterte *destroy*-Action:

Listing 1.20: ontrack/app/controllers/projects_controller.rb

```
def destroy
   @project = Project.find(params[:id])
   @project.destroy

   respond_to do |format|
     format.html { redirect_to projects_url }
     format.js   # default template destroy.rjs
```

```
      format.xml  { head :ok }
    end
  end
```

Die einzige Änderung gegenüber der Ursprungsversion ist der zusätzliche *format.js*-Eintrag im *respond_to*-Block. Da dem Eintrag kein expliziter Anweisungsblock folgt, reagiert Rails standardkonform und liefert ein RJS-Template mit Namen *destroy.rjs*, das wie folgt anzulegen ist:

Listing 1.21: ontrack/app/views/projects/destroy.rjs

```
page.remove "project_#{@project.id}"
```

Das Template löscht das Element mit der Id *project_ID* aus dem DOM-Tree des Browsers. Damit das für den *index*-View des *ProjectsController*s funktioniert, müssen die enthaltenen Tabellenzeilen um eine eindeutige Id erweitert werden:

Listing 1.22: ontrack/app/views/projects/index.rhtml

```
...
<% for project in @projects %>
  <tr id="project_<%= project.id %>">
```

Die hier beschriebene Erweiterung des *ProjectsController*s ist ein gutes Beispiel für die Einhaltung des DRY-Prinzips und die daraus resultierende geringere Codemenge in REST-Anwendungen. Eine einzige Zeile im Controller sorgt dafür, dass die selbe Action jetzt auch auf JavaScript-Requests reagieren kann.

Das Beispiel zeigt ausserdem eine generelle Regel für die Entwicklung von REST-Controllern auf: Je mehr Logik ausserhalb des *respond_to*-Blocks implementiert wird, desto weniger Wiederholungen enthält der resultierende Code.

1.15 Testen

Egal wie spannend REST-basierte Entwicklung mit Rails auch ist, das Testen sollte dabei nicht auf der Strecke bleiben. Dass wir eigentlich schon viel zu lange entwickelt haben, ohne dabei ein einziges Mal an unsere Unit-Tests zu denken, wird spätestens bei der Ausführung der Tests mit *rake* deutlich[8]:

```
> rake
...
Started
EEEEEEE.......
```

Die gute Nachricht ist: Alle Unit-Tests und der funktionale Test *ProjectsControllerTest* laufen noch. Die schlechte Nachricht ist: Alle sieben Tests des funktionalen *IterationsControllerTest* schlagen fehl.

Wenn alle Tests eines Testcases fehlschlagen, ist dies ein deutlicher Hinweis darauf, dass mit dem Test etwas grundsätzliches nicht stimmt. In unserem Fall liegt die Ursache für die Fehler auf der Hand. Der Testcase wurde vom Scaffold-Generator für

[8] Achtung: Anlegen der Testdatenbank *ontrack_test* nicht vergessen, sofern nicht bereits geschehen.

Iterationen ohne zugehöriges Projekt generiert. Da wir aber manuell eingegriffen haben, und Iterationen Projekten zugeordnet und den *IterationsController* entsprechend angepasst haben, ist klar, was der Grund für die fehlschlagenden Tests ist: Alle *IterationsController*-Actions erwarten die Id des ausgewählten Projekts im Request-Parameter *project_id*. Dieser Tatsache wird in keiner der Testmethoden Rechnung getragen.

Erweitern Sie die Request-Hash aller Testmethoden um den Parameter *project_id*. Wir greifen als Beispiel den Test *test_should_get_edit* auf:

Listing 1.23: ontrack/test/functional/iterations_controller_test.rb

```
def test_should_get_edit
  get :edit, :id => 1, :project_id => projects(:one)
  assert_response :success
end
```

Ausserdem muss zusätzlich das *projects*-Fixture vom *IterationsControllerTest* geladen werden:

```
fixtures :iterations, :projects
```

Nach diesen Änderungen sollten nur noch zwei Tests mit fehlschlagenden Assertions abbrechen: *test_should_create_iteration* und *test_should_update_iteration*. In beiden Fällen ist die Ursache eine fehlschlagende *assert_redirected_to*-Assertion:

```
assert_redirected_to iteration_path(assigns(:iteration))
```

Ganz klar was hier fehlt: Wir haben alle Redirects im *IterationsController* so angepasst, dass der erste Parameter jeweils die Projekt-Id ist. Die Assertion hingegen prüft ausschliesslich das Vorhandensein einer Iterations-Id im Redirect-Aufruf. In diesem Fall hat der Controller Recht und der Test muss angepasst werden:

```
assert_redirected_to iteration_path(projects(:one),
                                    assigns(:iteration))
```

Die Verwendung von Path-Methoden in Redirect-Assertions ist übrigens das einzige, was funktionale REST-Tests von funktionalen Tests in Nicht-REST-Anwendungen unterscheidet.

1.16 RESTful Clients: ActiveResource

In Zusammenhang mit REST ist häufig von ActiveResource die Rede. ActiveResource ist eine Rails-Bibliothek für die Erstellung von REST-basierten Webservice-Clients. Ein REST-basierter Webservice-Client kommuniziert über HTTP mit dem Server und verwendet dabei die vier REST-typischen HTTP-Verben.

ActiveResource ist kein Bestandteil von Rails 1.2, aber bereits im Development-Trunk verfügbar und kann via svn[9] installiert werden:

[9] http://subversion.tigris.org/

```
> cd ontrack/vendor
> mv rails rails-1.2
> svn co http://dev.rubyonrails.org/svn/rails/trunk rails
```

ActiveResource abstrahiert clientseitige Web-Ressourcen als Klassen, die von *ActiveResource::Base* erben. Als Beispiel verwenden wir die bereits existierende Server-Ressource *Project*, die wir clientseitig wie folgt modellieren:

```
require "activeresource/lib/active_resource"

class Project < ActiveResource::Base
  self.site = "http://localhost:3000"
end
```

Die ActiveResource-Bibliothek wird explizit importiert. Ausserdem wird über die Klassenvariable *site* die URL des Services spezifiziert. Die Klasse *Project* abstrahiert den clientseitigen Teil des Web Services so gut, dass beim Programmierer der Eindruck entsteht, er hätte es mit einer ganz normalen ActiveRecord-Klasse zu tun. Beispielsweise steht eine *find*-Methode zur Verfügung, die eine Ressource mit der angegebenen Id vom Server anfordert:

```
wunderloop = Project.find 1
puts wunderloop.name
```

Der find-Aufruf führt einen REST-konformen GET-Request durch:

```
GET /projects/1.xml
```

Der Server liefert die Antwort im XML-Format. Der Client erzeugt daraus ein ActiveResource-Objekt *wunderloop*, auf dem jetzt, ähnlich wie bei ActiveRecord-Modellen, Getter und Setter aufgerufen werden können. Wie sieht das ganze für Updates aus?

```
wunderloop.name = "Wunderloop Connect"
wunderloop.save
```

Der save-Aufruf konvertiert die Ressource nach XML und sendet sie via PUT an den Server:

```
PUT /projects/1.xml
```

Wechseln Sie in den Browser und laden Sie die Liste der Projekte neu. Das geänderte Projekt sollte jetzt einen neuen Namen haben.

Genau so einfach wie das Anfordern und Aktualisieren von Ressourcen ist deren Neuanlage via ActiveResource:

```
bellybutton = Project.new(:name => "Bellybutton")
bellybutton.save
```

Das neue Projekt wird via POST im XML-Format an den Server übertragen und dort gespeichert:

```
POST /projects.xml
```

Auch hier zeigt ein Reload im Browser das neue Projekt an. Bleibt noch die letzte der vier CRUD-Operationen, das Löschen von Ressourcen:

```
bellybutton.destroy
```

Der *destroy*-Aufruf wird per DELETE gesendet und führt serverseitig zum Löschen des Projekts:

```
DELETE /projects/2.xml
```

ActiveResource verwendet alle vier HTTP-Verben im Sinne von REST und stellt damit eine sehr gute clientseitige Abstraktion von REST-Ressourcen zur Verfügung. Aber auch andere von ActiveRecord bekannte Methoden funktionieren, wie z.B. die Suche nach allen Instanzen einer Ressource:

```
Project.find(:all).each do |p|
  puts p.name
end
```

Wir denken, dass ActiveResource eine sehr gute Grundlage für die Entwicklung von lose gekoppelten Systemen in Ruby bietet. Es lohnt sich also schon jetzt, einen Blick auf die im Trunk enthaltenen Basisklassen zu werfen und mit ihnen zu experimentieren.

1.17 Abschliessend

Es muss nicht alles REST sein. Hybridlösungen sind denkbar und problemlos umzusetzen. In der Regel fängt man bei Erscheinen neuer Rails-Features ja auch nicht auf der grünen Wiese an, sondern steckt mitten in einem Projekt. Es ist ohne weiteres möglich, einzelne Modelle und zugehörige Controller REST-basiert zu entwickeln und Erfahrungen damit zu sammeln. Bei komplett neuen Anwendungen sollte darüber nachgedacht werden, diese von vornherein RESTful zu entwickeln. Die Vorteile liegen auf der Hand: Klare Architektur, weniger Code und Multiclient-Fähigkeit.

Literaturverzeichnis

[1] *Ralf Wirdemann, Thomas Baustert*: Rapid Web Development mit Ruby on Rails, 2. Auflage, Hanser, 2007

[2] *Dave Thomas, David Heinemeier Hansson: Agile Web Development with Rails*, Second Edition, Pragmatic Bookshelf, 2006

[3] *Curt Hibbs*: Rolling with Ruby on Rails – Part 1,
http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html

[4] *Curt Hibbs*: Rolling with Ruby on Rails – Part 2,
http://www.onlamp.com/pub/a/onlamp/2005/03/03/rails.html

[5] *Amy Hoy*: Really Getting Started in Rails,
http://www.slash7.com/articles/2005/01/24/really-getting-started-in-rails.html

[6] *Roy Fielding*: Architectural Styles and the Design of Network-based Software Architectures,
http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm