Ruby on Rails 3.2

STEFAN WINTERMEYER

ADDISON-WESLEY

Installationsanleitungen
für OS X, Windows und Linux (Ubuntu und Debian)

Ruby on Rails-Features
Active Record, Scaffolding, REST, Routen, Cookies, Sessions, Action Mailer, Bundler, Gems, Asset Pipeline, Caching und mehr

Ruby-Grundlagen
Sprachelemente und -syntax, irb, Objektorientierung

Webserver-Szenarios für den produktiven Einsatz
mit Caching und Deployment

ALWAYS LEARNING

PEARSON

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

10 9 8 7 6 5 4 3 2 1

14 13

ISBN 978-3-8273-2989-9 Print; 978-3-86324-762-1 PDF; 978-3-86324-175-9 ePUB

ein Imprint der Pearson Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany

www.pearson.de
Umschlagkonzept: Kochan & Partner, München
Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Fachlektorat: Dominik Bamberger, @bamb00zzle
Korrektorat: Brigitte Hamerski, Willich
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: Reemers Publishing Services GmbH, Krefeld (www.reemers.de)
Druck: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

3 Erste Schritte mit Rails

3.1 Einleitung

Nachdem Sie sich mit Kapitel 2, Ruby-Grundlagen, mühsam in die Ruby-Grundlagen eingelesen haben, können wir jetzt spannender weitermachen. In diesem Kapitel starten wir ein erstes Rails-Projekt und arbeiten uns damit Stück für Stück in die Materie ein.

Auch in diesem Kapitel wird es manchmal hoppla-hopp zugehen. Wir stoßen auf typische Henne-Ei-Probleme.

3.1.1 Arbeits-Umgebung (Development)

Rails kennt drei verschiedene Arbeits-Umgebungen (Environments):

» Development

» Test

» Production

Wir arbeiten in diesem Kapitel nur mit der Development-Umgebung. Sobald Sie ein besseres Gefühl für Rails bekommen haben, beginnen wir mit Tests und benötigen dafür die entsprechende Umgebung (dort wird z. B. beim Start eines Tests die Test-Datenbank neu gefüllt und danach gelöscht). Später erkläre ich Ihnen dann die verschiedenen Szenarien, wie Sie Ihre Rails-Applikation aus der Development-Umgebung in die Production-Umgebung ausrollen können.

Die Development-Umgebung bringt bis auf einen Editor und einen Webbrowser alles mit, was Sie zum Entwickeln benötigen. So müssen Sie nicht extra einen Webserver installieren, sondern können den eingebauten Rails-Webserver benutzen. Der besticht nicht durch extreme Performance, aber das benötigen wir bei der Entwicklung ja auch nicht. Später kann man dann auf große Webserver wie Apache umsteigen. Das Gleiche gilt für die Datenbank.

Um in der Development-Umgebung zu arbeiten, müssen Sie erst mal nichts verändern – alle Befehle arbeiten per Default.

SQLite-3-Datenbank

Auch bei der Datenbank geht es in diesem Kapitel nicht um optimale Performance, sondern um einen einfachen Einstieg. Deshalb benutzen wir die SQLite-3-Datenbank. Dafür haben Sie bereits alles fertig installiert und müssen sich um nichts kümmern. Später erkläre ich Ihnen dann, wie Sie andere Datenbanken (z. B. MySQL) ansteuern können.

3.1.2 Warum alles auf Englisch?

Ganz tief im Herzen liebt Rails die englische Sprache. Das ist fast ein wenig ironisch, weil der Erfinder David Heinemeier Hansson (»DHH«) ja aus Dänemark stammt (er lebt und arbeitet heute in Chicago).

Rails' Liebe zur englischen Sprache muss man akzeptieren und sollte sogar versuchen, sie zu übernehmen. Vieles wird dadurch einfacher und logischer. Ein Großteil des Codes ist dann fast normal zu lesen. So verwenden sehr viele Mechanismen automagisch Plural oder Singular von englischen Wörtern. Wenn man sich damit anfreundet, Datenbank-Felder und -Tabellen mit englischen Begriffen zu benennen, dann kann man die ganze Macht dieser Magie ausnutzen. Diesen Mechanismus nennt man `Inflector`[1] oder *Inflections* (Beugungen/Flexionen[2]).

Im Buch verwende ich für Variablen, Klassen und Methoden englische Namen. Die Kommentare sind auf Deutsch geschrieben. Falls Sie bei internationalen Projekten mitmachen, sollten Sie logischerweise auch die Kommentare auf Englisch schreiben. Ja, ja, ... gut geschriebener Code braucht keine Kommentare. ;-)

3.2 Statische Inhalte (HTML- und Grafik-Dateien)

Jeder, der diesen Text liest, wird sich darüber im Klaren sein, dass man mit Rails irgendwie Webseiten ausliefern kann. Die Frage ist nur, wie. Legen wir erst mal ein neues Rails-Projekt an.

3.2.1 Rails-Projekt anlegen

Bevor wir hier überhaupt mit dem Allereinfachsten beginnen, überprüfen Sie bitte, ob Sie eine Ruby-Version 1.9.3 einsetzen:

```
MacBook:~ xyz$ ruby -v
ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-darwin11.3.0]
MacBook:~ xyz$
```

Und als Nächstes überprüfen wir, ob auch Rails 3.2 installiert ist:

```
MacBook:~ xyz$ rails -v
Rails 3.2.3
MacBook:~ xyz$
```

Das sieht gut aus. Falls Sie eine ältere Ruby- oder Rails-Version installiert haben, dann installieren Sie vor dem Weiterlesen die aktuelle Version (siehe Kapitel 1, Versionsauswahl und Installation).

Jetzt erstellen wir zuerst ein neues Rails-Projekt namens `testproject`. Da Ruby on Rails ein Framework ist, müssen wir als Erstes die entsprechende Verzeichnisstruktur und die Grundkonfiguration inkl. einiger Skripte einrichten. Das geht mit dem Befehl `rails new testproject` ratzfatz:

1 siehe `http://api.rubyonrails.org/classes/ActiveSupport/Inflector.html`
2 siehe `http://de.wikipedia.org/wiki/Flexion`

```
MacBook:~ xyz$ rails new testproject
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      create  app
      create  app/assets/images/rails.png
      create  app/assets/javascripts/application.js

[...]

Using rails (3.2.3)
Installing sass (3.1.16)
Installing sass-rails (3.2.5)
Installing sqlite3 (1.3.6) with native extensions
Installing uglifier (1.2.4)
Your bundle is complete! Use 'bundle show [gemname]' to see where a bundled gem is
   installed.
MacBook:~ xyz$
```

Bei früheren Rails-Versionen musste als Erstes ein `bundle install` ausgeführt werden. Ab Rails 3.2 wird dies aber automatisch beim Erstellen eines neuen Rails-Projektes gemacht. Damit stehen dem Rails-Projekt alle benötigten Gems zur Verfügung. Ein Gem ist eine Art Softwarebibliothek. Man kann damit bestimmte Funktionalitäten fix und fertig einbinden, ohne das Rad neu erfinden zu müssen.

Anschließend prüfen wir, ob die neue Rails-Applikation funktioniert. Dazu starten wir den mitgelieferten kleinen Webserver.

HINWEIS

Nein, keine Angst. Das ist lediglich der Webserver zum Entwickeln – für diesen Zweck ist er sehr praktisch.

HINWEIS

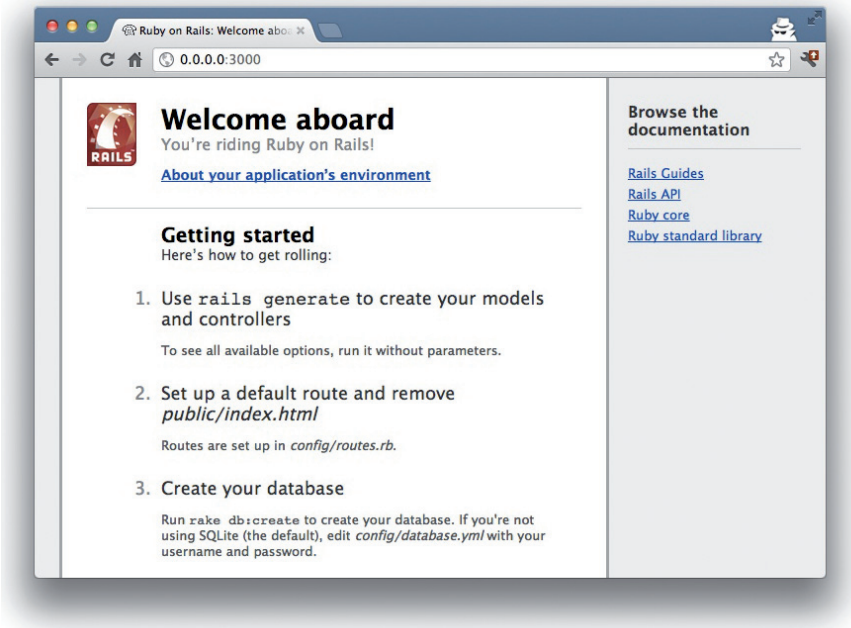
Bei verschiedenen Betriebssystemen (z. B. Mac OS X) erscheint beim ersten Starten einer Rails-Applikation – je nach Firewall-Einstellung – ein Fenster, das Sie fragt, ob die Firewall die entsprechende Verbindung erlauben soll. Da wir lokal arbeiten, können Sie das ruhigen Gewissens bejahen.

```
MacBook:~ xyz$ cd testproject
MacBook:testproject xyz$ rails server
=> Booting WEBrick
=> Rails 3.2.3 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-04-24 09:52:21] INFO  WEBrick 1.3.1
[2012-04-24 09:52:21] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.3.0]
[2012-04-24 09:52:21] INFO  WEBrick::HTTPServer#start: pid=57749 port=3000
```

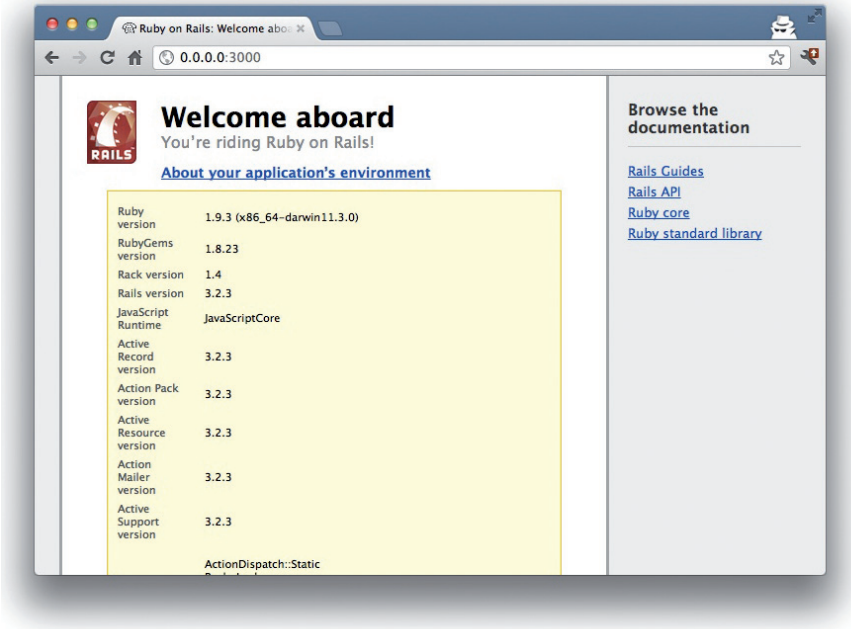
Der Start der Rails-Applikation sieht gut aus. Er sagt uns:

```
=> Rails 3.2.3 application starting in development on http://0.0.0.0:3000
```

Dann rufen wir die URL `http://0.0.0.0:3000` oder `http://localhost:3000` im Webbrowser auf.



Sieht gut aus. Rails scheint zu funktionieren. Wenn wir jetzt auf den Link »About your application's environment« klicken, erscheint eine Aufstellung der aktuellen Umgebung.



Gleichzeitig wird im Log des Webservers Folgendes angezeigt:

```
Started GET "/rails/info/properties" for 127.0.0.1 at 2012-04-24 09:59:53 +0200
Processing by Rails::InfoController#properties as */*
  Rendered inline template (2.5ms)
Completed 200 OK in 37ms (Views: 36.9ms | ActiveRecord: 0.0ms)
```

Da beim Aufruf der ersten Startseite kein entsprechender Eintrag erschien, muss es sich um zwei verschiedenartige Seiten handeln. Die erste Seite ist tatsächlich eine statische HTML-Seite, die unter `public/index.html` abgespeichert ist, und der Link, auf den wir geklickt haben führt ein Rails-Programm aus.

Mit Strg C können Sie den Webserver wieder stoppen.

3.2.2 Statische Seiten

Wie eben gesehen, gibt es bestimmte statische Seiten, Bilder und JavaScript-Dateien, die von Rails automatisch ausgeliefert werden. Erinnern wir uns noch mal an eine Teilausgabe vom Befehl `rails new testproject`:

```
MacBook:~ xyz$ rails new testproject
      create

      [...]

      create  public
      create  public/404.html
      create  public/422.html
      create  public/500.html
      create  public/favicon.ico
      create  public/index.html
      create  public/robots.txt

      [...]
```

Der Verzeichnisname `public` und die darin enthaltenen Dateien sehen schon sehr nach statischen Seiten aus. Probieren wir es einfach mal aus und legen die Datei `public/hello-world.html` mit folgendem Inhalt an:

```
<html>
<title>Hello World!</title>
<body>
  <h1>Hello World!</h1>
  <p>Eine Beispielseite.</p>
</body>
</html>
```

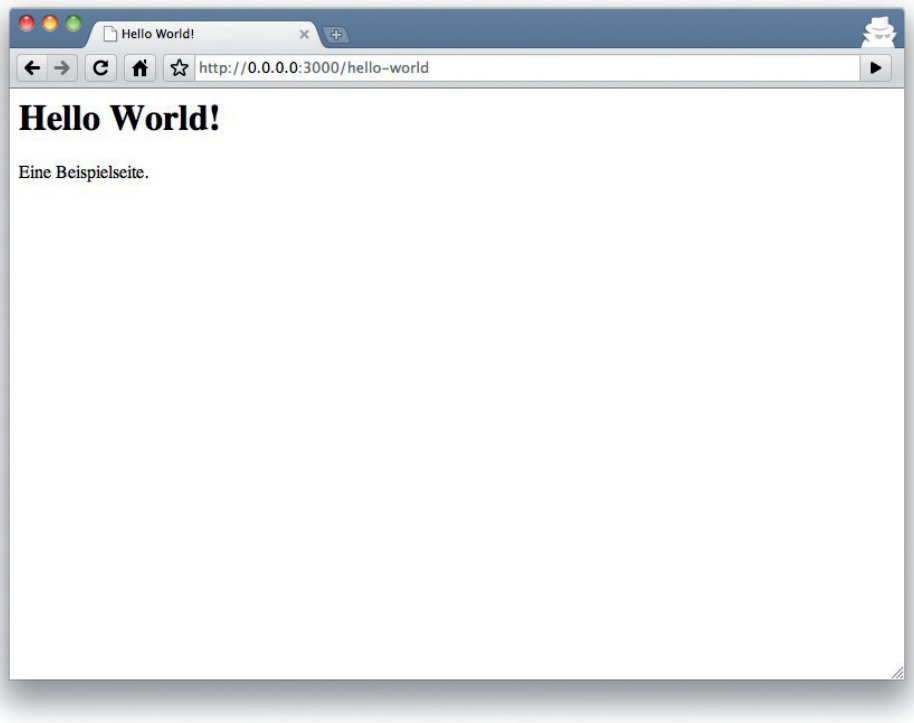
HINWEIS

Ich gehe davon aus, dass Sie rudimentäre HTML-Kenntnisse haben – für viel mehr reicht mein HTML-Wissen auch nicht ;-).

Jetzt noch den Rails-Webserver starten:

```
MacBook:testproject xyz$ rails server
=> Booting WEBrick
=> Rails 3.2.3 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-04-24 10:04:59] INFO  WEBrick 1.3.1
[2012-04-24 10:04:59] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.3.0]
[2012-04-24 10:04:59] INFO  WEBrick::HTTPServer#start: pid=57880 port=3000
```

Diese Webseite können wir uns unter der URL `http://0.0.0.0:3000/hello-world` anschauen:



Wir können natürlich auch die URL `http://0.0.0.0:3000/hello-world.html` nehmen. Allerdings sieht Rails HTML und damit die Datei-Endung `.html` quasi als Standard-Ausgabe-Format an und deshalb kann man sich das »`.html`« hier sparen.

So, jetzt wissen Sie bereits, wie man gänzlich statische Seiten in Rails integrieren kann. Das ist für Seiten praktisch, die sich nie ändern und die auch dann funktionieren sollen, wenn Rails gerade z. B. wegen eines Updates nicht läuft. Im Produktivbetrieb wird meistens vor dem Rails-Server noch ein klassischer Webserver wie Apache oder Nginx geschaltet. Dieser ist dann in der Lage, statische Dateien aus dem `public`-Verzeichnis selbstständig auszuliefern.

Mit Strg + C können Sie den Rails-Server wieder stoppen.

3.3 Dynamisch mit erb erzeugtes HTML

Kennen Sie PHP (ohne Frameworks)? Dann wird Ihnen der Inhalt einer `erb`-Datei sehr bekannt vorkommen. Es ist eine Mischung aus (beispielsweise) HTML und Ruby-Code. (`erb` steht für ***em**bedded **Ru**by*, also eingebettetes Ruby.) Allerdings können wir eine solche `erb`-Webseite nicht einfach in das Verzeichnis `public` legen, da dort abgelegte Seiten 1:1 ausgeliefert werden und nicht durch einen `erb`-Parser gehen. Dummerweise müssen wir dafür jetzt direkt mit dem MVC-Modell[3] anrücken. Wir brauchen einen Controller. Den können wir mit dem Befehl `rails generate controller` anlegen. Schauen wir uns mal die Hilfe an:

```
MacBook:testproject xyz$ rails generate controller
Usage:
  rails generate controller NAME [action action] [options]

Options:
      [--skip-namespace]        # Skip namespace (affects only isolated
                                  applications)
      [--old-style-hash]        # Force using old style hash (:foo => 'bar') on
                                  Ruby >= 1.9
  -e, [--template-engine=NAME]  # Template engine to be invoked
                                # Default: erb
  -t, [--test-framework=NAME]   # Test framework to be invoked
                                # Default: test_unit
      [--helper]                # Indicates when to generate helper
                                # Default: true
      [--assets]                # Indicates when to generate assets
                                # Default: true

Runtime options:
  -f, [--force]    # Overwrite files that already exist
  -p, [--pretend]  # Run but do not make any changes
  -q, [--quiet]    # Supress status output
  -s, [--skip]     # Skip files that already exist

Description:
    Stubs out a new controller and its views. Pass the controller name, either
    CamelCased or under_scored, and a list of views as arguments.

    To create a controller within a module, specify the controller name as a
    path like 'parent_module/controller_name'.

    This generates a controller class in app/controllers and invokes helper,
    template engine and test framework generators.

Example:
    'rails generate controller CreditCard open debit credit close'

    Credit card controller with URLs like /credit_card/debit.
        Controller:      app/controllers/credit_card_controller.rb
        Functional Test: test/functional/credit_card_controller_test.rb
        Views:           app/views/credit_card/debit.html.erb [...]
        Helper:          app/helpers/credit_card_helper.rb
MacBook:testproject xyz$
```

3 http://de.wikipedia.org/wiki/Model_View_Controller

Aha! Unten ist freundlicherweise direkt ein Beispiel angegeben:

```
rails generate controller CreditCard open debit credit close
```

Passt aber nicht direkt für unseren Fall.

Ich bin mutig und schlage vor, dass wir einfach Folgendes ausprobieren:

```
MacBook:testproject xyz$ rails generate controller Example test
      create  app/controllers/example_controller.rb
       route  get "example/test"
      invoke  erb
      create    app/views/example
      create    app/views/example/test.html.erb
      invoke  test_unit
      create    test/functional/example_controller_test.rb
      invoke  helper
      create    app/helpers/example_helper.rb
      invoke    test_unit
      create      test/unit/helpers/example_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/example.js.coffee
      invoke    scss
      create      app/assets/stylesheets/example.css.scss
MacBook:testproject xyz$
```

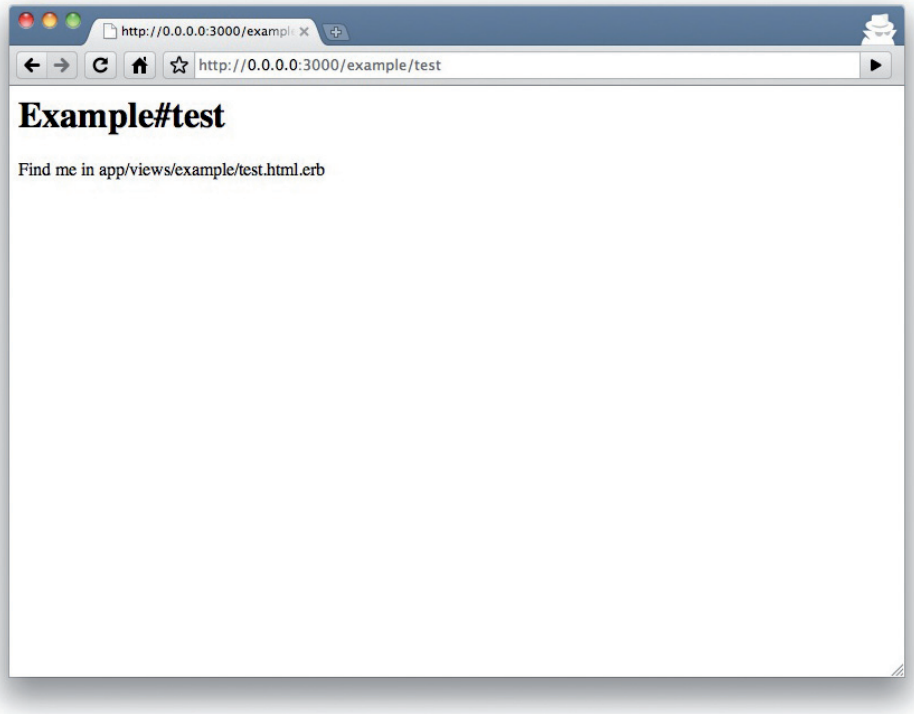
Puh, da wird ja direkt 'ne ganze Menge erstellt. Unter anderem eine Datei `app/views/example/test.html.erb`. Schauen wir uns diese nachfolgend an:

```
MacBook:testproject xyz$ cat app/views/example/test.html.erb
<h1>Example#test</h1>
<p>Find me in app/views/example/test.html.erb</p>
MacBook:testproject xyz$
```

Ist HTML, aber für eine valide HTML-Seite »fehlt« oben und unten etwas (der fehlende HTML-»Rest« wird in Abschnitt 3.3.2, Layouts erklärt). Zum Testen starten wir den Webserver

```
MacBook:testproject xyz$ rails server
=> Booting WEBrick
=> Rails 3.2.3 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-04-24 10:13:48] INFO  WEBrick 1.3.1
[2012-04-24 10:13:48] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.3.0]
[2012-04-24 10:13:48] INFO  WEBrick::HTTPServer#start: pid=57898 port=3000
```

und schauen uns die Webseite unter der URL `http://0.0.0.0:3000/example/test` im Browser an:



Im Log `log/development.log` finden wir den folgenden Eintrag:

```
Started GET "/example/test" for 127.0.0.1 at 2012-04-24 10:14:18 +0200
Processing by ExampleController#test as HTML
  Rendered example/test.html.erb within layouts/application (1.9ms)
Compiled example.css  (9ms)  (pid 57898)
Compiled application.css  (18ms)  (pid 57898)
Compiled jquery.js  (4ms)  (pid 57898)
Compiled jquery_ujs.js  (0ms)  (pid 57898)
Compiled example.js  (160ms)  (pid 57898)
Compiled application.js  (198ms)  (pid 57898)
Completed 200 OK in 305ms (Views: 304.2ms | ActiveRecord: 0.0ms)
```

Das ist fast schon verständlich geschrieben. Es kam also vom `localhost` (127.0.0.1) ein HTTP-GET-Request für die URI »`/example/test`« rein. Die wurde dann anscheinend vom Controller `ExampleController` mit der Methode `test` als HTML gerendert. Zusätzlich wurde noch ein Satz von CSS- und JavaScript-Dateien kompiliert (darauf gehen wir später im Buch ein). Das Ganze hat hier ungefähr 305 ms gedauert.

Jetzt müssen wir nur noch den Controller finden. Aber Sie haben Glück … ich weiß es nämlich. ;-) Alle Controller befinden sich im Verzeichnis `app/controllers`, und siehe da, dort ist auch tatsächlich die entsprechende Datei `app/controllers/example_controller.rb`.

```
MacBook:testproject xyz$ ls -l app/controllers/
total 16
-rw-r--r--  1 xyz  staff  80 Apr 24 09:43 application_controller.rb
-rw-r--r--  1 xyz  staff  69 Apr 24 10:11 example_controller.rb
MacBook:testproject xyz$
```

Öffnen Sie die Datei bitte mit Ihrem Lieblingseditor:

```
class ExampleController < ApplicationController
  def test
  end
end
```

Das ist sehr übersichtlich. Der Controller `ExampleController` stammt vom `ApplicationController` ab und enthält aktuell genau eine Methode namens `test`. Und diese Methode hat keinen Inhalt.

Sie werden sich fragen, woher Rails weiß, dass bei dem URL-Pfad `/example/test` der Controller `ExampleController` und die Methode `test` abzuarbeiten sind. Das wird nämlich nicht durch eine magische Logik, sondern durch eine einfache Routing-Konfiguration gesteuert. Diese finden Sie in der Datei `config/routes.rb` in der zweiten Zeile:

```
MacBook:testproject xyz$ cat config/routes.rb | grep example
  get "example/test"
MacBook:testproject xyz$
```

Diese Zeile wurde vom Befehl `rails generate controller` automatisch eingefügt. In der Routing-Datei können Sie auch beliebiges *Mapping* vornehmen. Aber dazu später mehr. Aktuell sehen unsere Routen sehr einfach aus. Mit dem Befehl `rake routes` können wir diese abfragen:

```
MacBook:testproject xyz$ rake routes
example_test GET /example/test(.:format) example#test
MacBook:testproject xyz$
```

Wir kümmern uns später noch genauer um die Routen (Kapitel 6, Routen (routes)). Ich wollte es an dieser Stelle nur nicht gänzlich überspringen.

INFO

Eine statische Datei im Verzeichnis `public` hat immer eine höhere Priorität als eine Route in der `config/routes.rb`! Wenn wir also eine statische Datei `public/example/test` abspeichern würden, würde die Route nicht mehr greifen.

3.3.1 Programmieren in einer erb-Datei

`erb`-Seiten können Ruby-Code enthalten. Damit kann programmiert werden, und damit können diese Seiten dynamischen Inhalt bekommen.

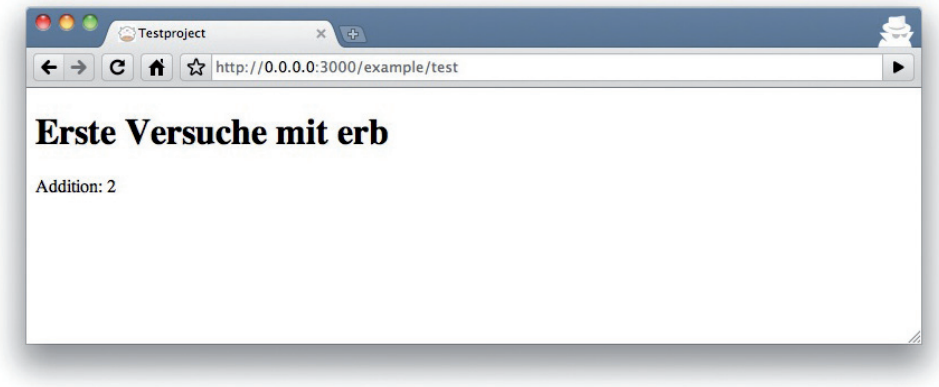
Fangen wir mit etwas ganz Einfachem an: der Addition von 1 und 1. Als Erstes probieren wir den Code im `irb` aus:

```
MacBook:testproject xyz$ irb
1.9.3p194 :001 > 1 + 1
 => 2
1.9.3p194 :002 > exit
MacBook:testproject xyz$
```

Das war einfach. Die `erb`-Datei `app/views/example/test.html.erb` füllen wir wie folgt:

```
<h1>Erste Versuche mit erb</h1>
<p>Addition:
<%= 1 + 1 %>
</p>
```

Danach mit `rails server` den Webserver starten (falls noch nicht getan) und per Browser auf die Seite gehen:



Ruby-Code, dessen Ergebnis ausgegeben werden soll, wird von einem `<%=` und einem `%>` eingeschlossen. Es können nur Strings ausgegeben werden.

Jetzt werden Sie sich vielleicht fragen: Wie kann denn das Ergebnis einer Addition von zwei Fixnums als Text angezeigt werden? Schauen wir erst mal im `irb` nach, ob es wirklich ein Fixnum ist:

```
MacBook:testproject xyz$ irb
1.9.3p194 :001 > 1.class
 => Fixnum
1.9.3p194 :002 > (1 + 1).class
 => Fixnum
1.9.3p194 :003 > exit
MacBook:testproject xyz$
```

Ja, sowohl die Zahl 1 also auch das Ergebnis von 1 + 1 ist ein Fixnum. Was ist passiert? Rails ist so intelligent, alle Objekte in einem View (das ist die Datei `test.html.erb`), die nicht bereits ein String sind, automatisch mit der Methode `.to_s` aufzurufen, welche per Konvention immer den Inhalt des Objektes in einen String konvertiert. Noch mal kurz ins `irb`:

```
MacBook:testproject xyz$ irb
1.9.3p194 :001 > (1 + 1)
 => 2
1.9.3p194 :002 > (1 + 1).class
 => Fixnum
1.9.3p194 :003 > (1 + 1).to_s
 => "2"
1.9.3p194 :004 > (1 + 1).to_s.class
 => String
1.9.3p194 :005 > exit
MacBook:testproject xyz$
```

Das mit dem Ruby-Code schauen wir uns jetzt genauer an. In einer `.html.erb`-Datei gibt es zusätzlich zu den HTML-Elementen zwei Arten von Ruby-Code-Anweisungen:

» `<% ... %>`

Führt den enthaltenen Ruby-Code aus, aber gibt nichts aus (außer Sie verwenden explizit so etwas wie `print` oder `puts`).

» `<%= ... %>`

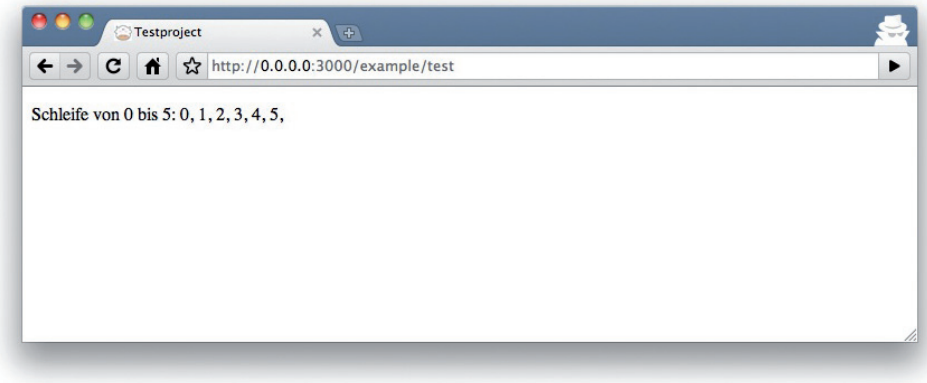
Führt den enthaltenen Ruby-Code aus und gibt das Ergebnis als Text aus. Dabei werden seit Rails 3.0 automatisch bestimmte Zeichen »escapt«. Falls Sie einmal nicht escapten Text ausgeben möchten, so müssen Sie das mit `raw(string)` realisieren.

Sofern also ein Objekt eine Methode `.to_s` hat oder das Objekt selber schon ein String ist, kann man es als Ergebnis im View innerhalb einer `<%= ... %>` Kapselung ausgeben.

Um ganz sicher zu sein, noch ein Beispiel. Wir ändern die `app/views/example/test.html.erb` wie folgt:

```
<p>Schleife von 0 bis 5:
<% (0..5).each do |i| %>
<%= "#{i}, " %>
<% end %>
</p>
```

Das sieht dann im Browser so aus:



Schauen wir uns nachfolgend den HTML-Source-Code (-Quelltext) im Browser an:

```
<!DOCTYPE html>
<html>
<head>
  <title>Testproject</title>
  <link href="/assets/application.css?body=1" media="all" rel="stylesheet"
   type="text/css" />
<link href="/assets/example.css?body=1" media="all" rel="stylesheet" type="text/
   css" />
  <script src="/assets/jquery.js?body=1" type="text/javascript"></script>
<script src="/assets/jquery_ujs.js?body=1" type="text/javascript"></script>
<script src="/assets/example.js?body=1" type="text/javascript"></script>
<script src="/assets/application.js?body=1" type="text/javascript"></script>
  <meta content="authenticity_token" name="csrf-param" />
<meta content="TDIallkBmGrMzsL6TC8Chet4r/X1yLKOtthFmIig4+E=" name="csrf-token" />
</head>
<body>

<p>Schleife von 0 bis 5:
0,
1,
2,
3,
4,
5,
</p>


</body>
</html>
```

Alles klar? Es gibt zwei mögliche offene Fragen:

1. Ich verstehe gar nichts. Mit dem Ruby-Code komme ich nicht zurecht. Können Sie das noch mal erklären?

   Kann es sein, dass Sie Kapitel 2, Ruby-Grundlagen, nicht komplett durchgearbeitet haben? Bitte nehmen Sie sich die Zeit dafür. Sonst macht hier das alles keinen Sinn.

2. Ich verstehe den Ruby-Code und die HTML-Ausgabe. Allerdings verstehe ich nicht, warum drum herum noch HTML-Code gerendert wurde, den ich gar nicht geschrieben habe. Woher kommt der, und kann ich ihn beeinflussen?

   Sehr gute Frage! Dazu kommen wir sofort (siehe Abschnitt 3.3.2, Layouts).

Die Feinheiten von `erb` werden Sie jetzt Stück für Stück erlernen. Es handelt sich dabei nicht um Zauberei.

3.3.2 Layouts

Die `erb`-Datei im Verzeichnis `app/views/example/` bildet nur den Kern der späteren HTML-Seite. Per Default wird immer eine automatisch generierte `app/views/layouts/application.html.erb` drum herum gerendert. Schauen wir uns die mal an:

```
MacBook:testproject xyz$ cat app/views/layouts/application.html.erb
<!DOCTYPE html>
<html>
<head>
  <title>Testproject</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
MacBook:testproject xyz$
```

Ich löse das Rätsel auf: Interessant ist die Zeile:

```
<%= yield %>
```

Mit `<%= yield %>` wird hier die View-Datei inkludiert. Die drei Zeilen mit den Stylesheets und dem JavaScript lassen wir erst mal so, wie sie sind. Damit werden default CSS- und JavaScript-Dateien eingebaut.

Die Datei `app/views/layouts/application.html.erb` bietet Ihnen die Möglichkeit, das Grund-Layout für die gesamte Rails-Applikation festzulegen. Wenn Sie als Header für jede Seite ein `<hr>` und darüber einen Text eintragen wollen, dann können Sie das zwischen dem `<%= yield %>-` und dem `<body>`-Tag machen:

```
<!DOCTYPE html>
<html>
<head>
  <title>Testproject</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<p>Ein Test.</p>
<hr>

<%= yield %>

</body>
</html>
MacBook:testproject xyz$
```

Sie können aber auch im Verzeichnis `app/views/layouts/` noch andere Layouts anlegen und diese je nach Situation anwenden, aber lassen wir das erst mal. Wichtig ist, dass Sie die Grundidee verstehen.

3.3.3 Instanz-Variablen vom Controller zum View übergeben

Einer der Todsünden im MVC-Modell[4] ist ja bekanntlich, zu viel Programm-Logik im View unterzubringen. Quasi so, wie man früher oft in PHP programmiert hat. Bei MVC ist eins der Ziele, dass jeder beliebige HTML-Designer einen View erstellen kann, ohne sich über die Programmierung Gedanken machen zu müssen. Ja, ja, ... wenn das immer so einfach wäre. Trotzdem gehen wir das mal gedanklich weiter: Wenn ich im Controller einen Wert habe, den ich im View anzeigen will, so benötige ich dafür einen Mechanismus. Dieser heißt Instanz-Variable (*instance variable*) und fängt immer mit einem `@` an. Wer sich nicht mehr 100 % sicher ist, welche Variable welchen Geltungsbereich (*Scope*) hat, der sollte ganz fix noch mal einen Blick in Abschnitt 2.6.5, Gültigkeitsbereich (*Scope*) von Variablen, werfen.

Im folgenden Beispiel fügen wir im Controller eine Instanz-Variable für die aktuelle Uhrzeit ein und fügen diese dann im View ein. Wir nehmen also Programmier-Intelligenz aus dem View in den Controller.

Die Controller Datei `app/controllers/example_controller.rb` sieht so aus:

```
class ExampleController < ApplicationController
  def test
    @current_time = Time.now
  end

end
```

In der View-Datei `app/views/example/test.html.erb` können wir dann auf diese Instance-Variable zurückgreifen:

```
<p>
Die aktuelle Uhrzeit ist
<%= @current_time %>
</p>
```

Wir haben jetzt eine klare Trennung von Programmierlogik und Darstellungslogik mit dem Controller und dem View. So können wir im Controller die Uhrzeit je nach Zeitzone des Anwenders automatisch anpassen, ohne dass sich der Designer der Seite darum kümmern muss. Wie immer wird im View automatisch die Methode `to_s` angewendet.

Mir ist klar, dass jetzt keiner aufspringen und schreien wird: »Danke für die Erleuchtung! Ich werde nur noch sauber nach MVC programmieren.« Das obige Beispiel ist der erste kleine Schritt in die Richtung und zeigt, wie wir einfach mit Instanz-Variablen Werte aus dem Controller in den View bringen können.

3.3.4 Partials

Selbst bei kleinen Webprojekten gibt es oft wiederkehrende Elemente. Das kann zum Beispiel ein Footer der Seite mit den Kontaktdaten sein oder ein Menü. Rails gibt uns die Möglichkeit, diesen HTML-Code in sogenannte »*Partials*« abzuspeichern und dann innerhalb eines Views einzubin-

4 http://de.wikipedia.org/wiki/Model_View_Controller

den. Ein Partial wird ebenfalls im `app/views/example/` Verzeichnis abgespeichert. Allerdings muss der Dateiname mit einem Unterstrich (*Underscore* = _) anfangen.

HINWEIS

Das englische Adjektiv partial heißt so viel wie Teil-..., partiell oder unvollständig. Partials sind also so etwas wie Teile, Stückchen oder Vorlagen-Schnipsel.

Als Beispiel fügen wir unserer Seite jetzt einen Mini-Footer in einem eigenen Partial hinzu. Dafür schreiben wir in die neue Datei `app/views/example/_footer.html.erb` den folgenden Inhalt:

```
<hr>
<p>
Copyright 2009 - <%= Date.today.year %> beim Osterhasen
</p>
```

Die Datei `app/views/example/test.html.erb` verändern wir wie folgt und fügen mit dem Befehl `render` das Partial ein:

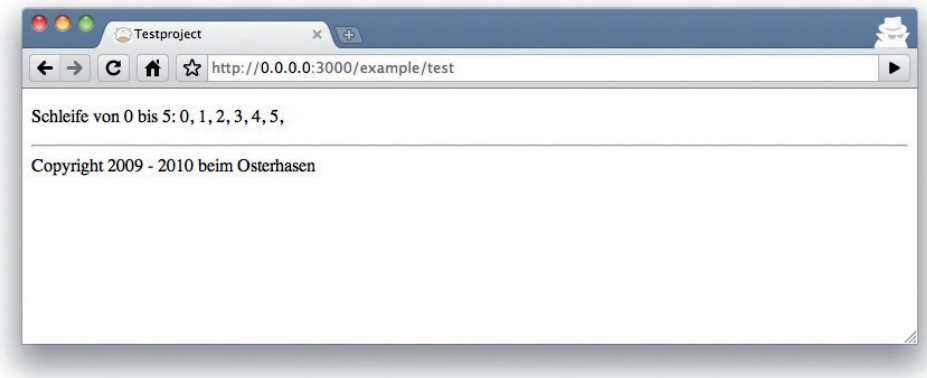
```
<p>Schleife von 0 bis 5:
<% (0..5).each do |i| %>
<%= "#{i}, " %>
<% end %>
</p>

<%= render "footer" %>
```

Es sind also jetzt folgende Dateien im Verzeichnis `app/views/example`:

```
MacBook:testproject xyz$ ls -l app/views/example
total 8
-rw-r--r--  1 xyz  staff   0 Apr 24 11:03 _footer.html.erb
-rw-r--r--  1 xyz  staff  81 Apr 24 10:33 test.html.erb
MacBook:testproject xyz$
```

Die neue Webseite sieht nun so aus:



INFO

Der Name eines Partials im Code wird immer *ohne* den Unterstrich (_) am Anfang und *ohne* die `.erb-` und `.html`-Endung angegeben. Aber die wirkliche Datei muss im Dateinamen mit einem Unterstrich anfangen und auch am Ende mit der `.erb`- und `.html`-Endung aufhören.

Partials können auch aus anderen Bereichen des Unterverzeichnisses `app/views` eingebunden werden. So können Sie für wiederkehrende und übergreifende Inhalte beispielsweise ein Verzeichnis `app/views/shared` einrichten und dort eine Datei `_footer.html.erb` anlegen. Das Einbetten im `erb`-Code würde dann mit folgender Zeile erfolgen:

```
<%= render "shared/footer" %>
```

HINWEIS

Das Footer-Problem würde man – je nach Programmiergeschmack – in einem richtigen Projekt nicht mit einem Partial lösen, das überall lokal aufgerufen wird, sondern eher zentral in der `app/views/layouts/application.html.erb`.

Variablen an ein Partial übergeben

Partials sind im DRY (***D**on't **R**epeat **Y**ourself*)-Gedanken sehr gut. Aber was sie erst richtig praktisch macht, ist die Möglichkeit, Variablen zu übergeben. Bleiben wir bei unserem Copyright-Beispiel. Wenn wir das Startjahr als Wert übergeben möchten, so können wir das mit folgender Erweiterung in der Datei `app/views/example/_footer.html.erb` einbauen:

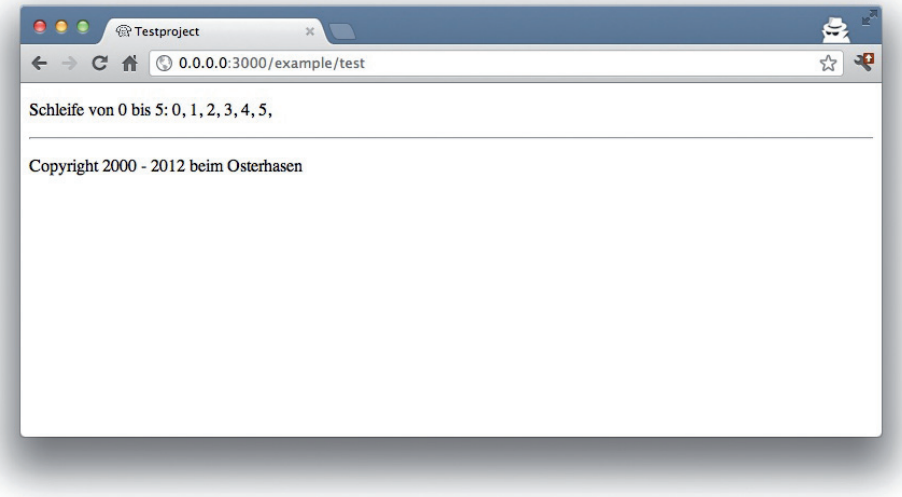
```
<hr />
<p>
Copyright <%= start_year %> - <%= Date.today.year %> beim Osterhasen
</p>
```

Ändern wir dazu die `app/views/example/test.html.erb` wie folgt:

```
<p>Schleife von 0 bis 5:
<% (0..5).each do |i| %>
<%= "#{i}, " %>
<% end %>
</p>

<%= render 'footer', :start_year => '2000' %>
```

Wenn wir jetzt die URL `http://0.0.0.0:3000/example/test` aufrufen, so sehen wir die 2000:



Manchmal braucht man ein Partial, das teilweise eine Local-Variable benutzt und an anderer Stelle braucht man das gleiche Partial, aber ohne Local-Variable. Das können wir im Partial selber mit einer if-Abfrage abfangen:

```
<hr />
<p>
Copyright
<% if defined? start_year %>
<%= start_year %>
 -
<% end %>
<%= Date.today.year %> beim Osterhasen
</p>
```

HINWEIS

Mit `defined?` wird in Ruby überprüft, ob eine Expression definiert ist.

Dieses Partial könnte man mit `<%= render 'footer', :start_year => '2000' %>` und mit `<%= render 'footer' %>` aufrufen.

Sie sehen ebenfalls, dass ich beim Einbinden eines einfaches Partials eine kürzere Schreibweise benutzen kann, als bei der Version mit `locals`.

Andere Schreibweise

In Abschnitt 3.3.4, Partials benutzen wir nur die Kurzform, um Partials zu rendern. Oft sehen Sie auch diese Langform:

```
<%= render :partial => "footer", :locals => { :start_year => '2000' } %>
```

Weitere Dokumentation zum Thema Partials

Wir haben hier wirklich nur die Oberfläche angekratzt. Partials sind sehr mächtige Werkzeuge. Unter `http://guides.rubyonrails.org/layouts_and_rendering.html#using-partials` finden Sie die Doku von Ruby on Rails zum Thema Partials.

3.4 Redirects (Umleitungen)

Redirects sind Befehle, mit denen Sie innerhalb des Controllers auf andere Methoden oder auch auf ganz andere Webseiten »springen«, also weiterleiten, können.

HINWEIS

Ein redirect gibt eine »302 Moved« response mit dem neuen Ziel an den Browser zurück.

Legen wir ein neues Rails-Projekt für ein entsprechendes Beispiel an:

```
MacBook:~ xyz$ rails new redirect_example
      create
      create  README.rdoc
      create  Rakefile

[...]

Using sqlite3 (1.3.6)
Using uglifier (1.2.4)
Your bundle is complete! Use 'bundle show [gemname]' to see where a bundled gem is
   installed.
MacBook:~ xyz$ cd redirect_example
MacBook:redirect_example xyz$
```

Um zu springen, brauchen wir einen Controller mit mindestens zwei verschiedenen Methoden. Und auf gehts:

```
MacBook:redirect_example xyz$ rails generate controller Game ping pong
      create  app/controllers/game_controller.rb
       route  get "game/pong"
       route  get "game/ping"
      invoke  erb
      create    app/views/game
      create    app/views/game/ping.html.erb
      create    app/views/game/pong.html.erb
      invoke  test_unit
      create    test/functional/game_controller_test.rb
      invoke  helper
      create    app/helpers/game_helper.rb
      invoke    test_unit
      create      test/unit/helpers/game_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/game.js.coffee
      invoke    scss
      create      app/assets/stylesheets/game.css.scss
MacBook:redirect_example xyz$
```

Starten wir mal den Rails-Server und rufen mit dem Browser erst `http://0.0.0.0:3000/game/ping` und dann `http://0.0.0.0:3000/game/pong` auf:

```
MacBook:redirect_example xyz$ rails server
=> Booting WEBrick
=> Rails 3.2.3 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-04-24 13:59:36] INFO  WEBrick 1.3.1
[2012-04-24 13:59:36] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.3.0]
[2012-04-24 13:59:36] INFO  WEBrick::HTTPServer#start: pid=58476 port=3000


Started GET "/game/ping" for 127.0.0.1 at 2012-04-24 14:00:01 +0200
Processing by GameController#ping as HTML
  Rendered game/ping.html.erb within layouts/application (1.9ms)
Compiled game.css  (8ms)  (pid 58476)
Compiled application.css  (18ms)  (pid 58476)
Compiled jquery.js  (3ms)  (pid 58476)
Compiled jquery_ujs.js  (0ms)  (pid 58476)
Compiled game.js  (139ms)  (pid 58476)
Compiled application.js  (173ms)  (pid 58476)
Completed 200 OK in 271ms (Views: 270.4ms | ActiveRecord: 0.0ms)

[...]

Started GET "/game/pong" for 127.0.0.1 at 2012-04-24 14:00:07 +0200
Processing by GameController#pong as HTML
  Rendered game/pong.html.erb within layouts/application (1.2ms)
Completed 200 OK in 8ms (Views: 7.8ms | ActiveRecord: 0.0ms)
```

Alles ganz normal. Das Projekt verfügt über zwei mögliche Routen:

```
MacBook:redirect_example xyz$ rake routes
game_ping GET /game/ping(.:format) game#ping
game_pong GET /game/pong(.:format) game#pong
MacBook:redirect_example xyz$
```

Der Controller `app/controllers/game_controller.rb` hat folgenden Inhalt:

```
MacBook:redirect_example xyz$ cat app/controllers/game_controller.rb
class GameController < ApplicationController
  def ping
  end

  def pong
  end
end
MacBook:redirect_example xyz$
```

Jetzt zum Redirect: Wie können wir es erreichen, dass wir beim Aufruf von `http://0.0.0.0:3000/game/ping` direkt auf die Methode `pong` weitergeleitet werden? Einfach, werden Sie sagen, indem wir die Route in der `config/routes.rb` ändern. Da haben Sie Recht. Dafür brauchen wir also nicht zwingend ein Redirect. Wenn wir aber vor der Umleitung in der `ping`-Methode noch etwas

abarbeiten wollen, dann geht das nur mit einem `redirect_to` im Controller `app/controllers/game_controller.rb`:

```
class GameController < ApplicationController
  def ping
   logger.info '++++++++++  Beispiel  ++++++++++'
   redirect_to game_pong_path
  end

  def pong
  end
end
```

Beim Aufruf von `http://0.0.0.0:3000/game/ping` landen wir nach dem automatischen Redirect auf `http://0.0.0.0:3000/game/pong` und sehen in der Log-Ausgabe die Logger-Ausgabe:

```
Started GET "/game/ping" for 127.0.0.1 at 2012-04-24 14:24:08 +0200
Processing by GameController#ping as HTML
++++++++++  Beispiel  ++++++++++
Redirected to http://0.0.0.0:3000/game/pong
Completed 302 Found in 1ms (ActiveRecord: 0.0ms)


Started GET "/game/pong" for 127.0.0.1 at 2012-04-24 14:24:08 +0200
Processing by GameController#pong as HTML
  Rendered game/pong.html.erb within layouts/application (0.3ms)
Completed 200 OK in 7ms (Views: 6.7ms | ActiveRecord: 0.0ms)
```

Aber was ist `game_pong_path`? Schauen wir uns dazu die für diese Rails-Applikation generierten Routen an:

```
MacBook:redirect_example xyz$ rake routes
game_ping GET /game/ping(.:format) game#ping
game_pong GET /game/pong(.:format) game#pong
MacBook:redirect_example xyz$
```

Sie sehen, dass die Route zur Action `ping` des Controllers `GameController` den Namen `game_ping` bekommen hat (siehe Anfang der Zeile). Wir könnten den Redirect auch folgendermaßen schreiben:

```
redirect_to :action => 'pong'
```

Auf die Details und einzelnen Möglichkeiten des Redirects gehen wir später im jeweils konkreten Fall ein. Nur schon so viel vorweg: Man kann nicht nur auf eine andere Methode, sondern auch auf einen anderen Controller oder eine ganz andere Webseite redirecten.

3.5 Flash-Meldungen (Flash messages)

Den Begriff »Flash messages« oder »Flash-Meldungen« halte ich für denkbar ungeeignet gewählt. Mit dem »Flash« assoziiert fast jeder mehr oder weniger bunte Webseiten, die mit dem Adobe-Shockwave-Flash-Plug-in realisiert wurden. Aber in Rails sind Flash-Nachrichten etwas

ganz anderes. Das sind Meldungen, die z. B. nach einem Redirect (siehe Abschnitt 3.4, Redirects (Umleitungen)) auf der neuen Seite angezeigt werden.

Flash-Nachrichten sind quasi gute Freunde von Redirects. Nicht selten arbeiten beide im Team, um dem User Feedback über eine gerade vollzogene Aktion zu geben. Ein typisches Beispiel einer Flash-Meldung ist das Feedback des Systems, wenn ein User sich eingeloggt hat. Dann wird er oft wieder auf die ursprüngliche Seite »redirectet« und bekommt zusätzlich noch das Feedback: »Sie sind jetzt eingeloggt.«

HINWEIS

Der englische Begriff Flash message lässt sich hier halbwegs gut mit *Einblendung/eingeblendete Meldung* übersetzen.

Wir bauen als Beispiel noch mal das Ping-Pong-Szenario aus Abschnitt 3.4, Redirects (Umleitungen) auf:

```
MacBook:~ xyz$ rails new pingpong

      [...]

MacBook:~ xyz$ cd pingpong
MacBook:pingpong xyz$ rails generate controller Game ping pong

      [...]

MacBook:pingpong xyz$
```

Die `app/controllers/game_controller.rb` füllen wir mit folgendem Inhalt:

```
class GameController < ApplicationController
  def ping
   redirect_to game_pong_path, notice: 'Ping-Pong!'
  end

  def pong
  end
end
```

Jetzt starten wir den Rails-Webserver mit `rails server` und gehen per Browser auf `http://0.0.0.0:3000/game/ping`. Wir werden von `ping` auf `pong` umgeleitet. Von der Flash-Nachricht »Ping-Pong!« ist aber nichts zu sehen. Dazu müssen wir `app/views/layouts/application.html.erb` erweitern:

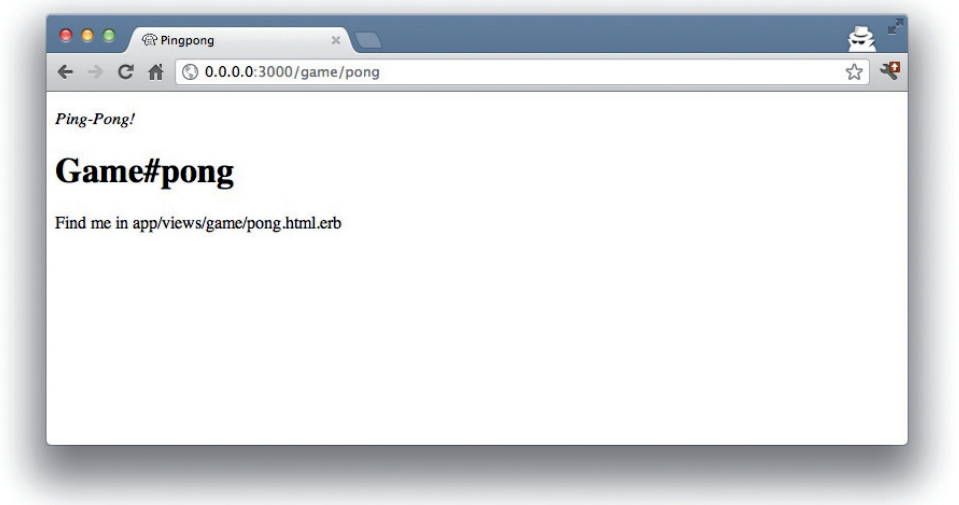
```
<!DOCTYPE html>
<html>
<head>
  <title>Pingpong</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>
```

```
<%- flash.each do |name, message| %>
<p><i><%= "#{name}: #{message}" %></i></p>
<% end %>

<%= yield %>

</body>
</html>
```

Erst jetzt sehen wir beim Aufruf von `http://0.0.0.0:3000/game/ping` im Browser die Flash-Nachricht am Kopf der Seite:



Beim Aufruf von `http://0.0.0.0:3000/game/pong` bekommen wir weiterhin normal die Pong-Seite angezeigt. Beim Aufruf von `http://0.0.0.0:3000/game/ping`[5] werden wir aber zur Pong-Seite redirectet und bekommen oben die Flash-Nachricht angezeigt:

TIPP

Wenn Sie später mal eine Flash-Nachricht nicht angezeigt bekommen, dann überprüfen Sie zuerst im View, ob dort die Flash-Nachricht überhaupt ausgegeben wird.

3.5.1 Verschiedene Arten von Flash-Meldungen

Flash-Nachrichten werden automagisch in einem Hash an den View übergeben. Per Default gibt es dabei die Arten `error` (Fehler), `warning` (Warnung) und `notice` (Notiz, Hinweis). Allerdings kann man auch selber eine Kategorie erfinden und diese später im View abfragen.

5 `http://0.0.0.0:3000/game/pong`

3.5.2 Verschiedene Syntaxen

Je nach Vorliebe eines Programmierers, werden Sie in der Praxis auf verschiedene Syntaxen von Flash-Nachrichten treffen. Ich will mich nicht an der Diskussion über die beste Syntax beteiligen, aber ich möchte Ihnen kurz die zwei häufigsten Varianten vorstellen:

```
» redirect_to game_pong_path, notice: 'Ping-Pong!'
» flash[:notice] = 'Ping-Pong!'
  redirect_to action: pong
```

3.5.3 Warum gibt es überhaupt Flash-Nachrichten?

Sie werden sich vielleicht fragen: »Warum gibt es Flash-Nachrichten? Das kann ich mir ja auch selber bauen, wenn ich es brauche.« Stimmt. Flash-Nachrichten haben hauptsächlich den Vorteil, dass sie einen definierten und für alle Programmierer gleichen Weg zur Verfügung stellen. So muss man nicht das Rad neu erfinden.

3.6 Die Rails-Konsole (Console)

Die Console in Rails ist nichts weiter als ein um die Rails-Umgebung aufgebohrtes `irb` (siehe Abschnitt 2.4, irb). Sowohl beim Entwickeln als auch bei der Administration ist die Console sehr praktisch, da die komplette Rails-Umgebung abgebildet wird und zur Verfügung steht. So kann man leicht mal eine Methode ausprobieren, ohne dafür direkt ein eigenes Programm zu schreiben.

Wir bauen als Arbeitsumgebung wieder das Ping-Pong-Szenario aus Abschnitt 3.4, Redirects (Umleitungen) auf:

```
MacBook:~ xyz$ rails new pingpong

      [...]

MacBook:~ xyz$ cd pingpong
MacBook:pingpong xyz$ rails generate controller Game ping pong

      [...]

MacBook:pingpong xyz$
```

Die Rails-Console wird mit dem Befehl `rails console` gestartet:

```
MacBook:pingpong xyz$ rails console
Loading development environment (Rails 3.2.3)
1.9.3p194 :001 >
```

Und mit `exit` kommt man wieder raus:

```
1.9.3p194 :001 > exit
MacBook:pingpong xyz$
```

In der Console hat man Zugriff auf alle Variablen, die man auch später in der richtigen Applikation hat:

```
MacBook:pingpong xyz$ rails console
Loading development environment (Rails 3.2.3)
1.9.3p194 :001 > Rails.env
 => "development"
1.9.3p194 :002 > Rails.root
 => #<Pathname:/Users/xyz/pingpong>
1.9.3p194 :003 > exit
MacBook:pingpong xyz$
```

Ganz praktisch ist auch `app`, um verschiedene Routing-Sachen zu analysieren:

```
MacBook:pingpong xyz$ rails console
Loading development environment (Rails 3.2.3)
1.9.3p194 :001 > app.url_for(:controller => 'game', :action => 'ping')
 => "http://www.example.com/game/ping"
1.9.3p194 :002 > app.get 'game/ping'
 => 200
1.9.3p194 :003 > app.get 'game/gibt_es_nicht'
 => 404
1.9.3p194 :004 > exit
MacBook:pingpong xyz$
```

Schon in Kapitel 4, ActiveRecord, werden Sie viel mit der Console arbeiten und die Möglichkeiten schätzen lernen.

3.7 Was ist ein Generator?

Wir haben in diesem Kapitel den Befehl `rails generate controller` benutzt. Damit starten wir den Generator namens `controller`. Es gibt noch andere Generatoren. Eine Liste der verfügbaren Generatoren bekommen Sie mit `rails generate` angezeigt:

```
MacBook:pingpong xyz$ rails generate
Usage: rails generate GENERATOR [args] [options]

General options:
  -h, [--help]     # Print generator's options and usage
  -p, [--pretend]  # Run but do not make any changes
  -f, [--force]    # Overwrite files that already exist
  -s, [--skip]     # Skip files that already exist
  -q, [--quiet]    # Suppress status output

Please choose a generator below.

Rails:
  assets
  controller
  generator
  helper
  integration_test
  mailer
  migration
  model
```

```
  observer
  performance_test
  resource
  scaffold
  scaffold_controller
  session_migration
  task

Coffee:
  coffee:assets

Jquery:
  jquery:install

Js:
  js:assets

MacBook:pingpong xyz$
```

Was macht ein Generator? Ein Generator nimmt dem Programmierer stupide Arbeit ab. Er legt Dateien an und füllt sie mit Inhalt abhängig von den übergebenen Parametern. Das Gleiche können Sie auch händisch ohne den Generator machen. Sie müssen also nicht einen Generator benutzen. Er soll in erster Linie eine Arbeitserleichterung sein und Fehlerquellen bei stupiden Aufgaben vermeiden.

3.8 Misc

Sie haben jetzt schon mal eine einfache Rails-Applikation ausgeführt und werden im nächsten Kapitel tief ins Thema ActiveRecord einsteigen. Deshalb an dieser Stelle eine minimale Erklärung zu einigen Begrifflichkeiten, die in der Rails-Welt immer mal wieder auftauchen.

3.8.1 »Model View Controller«-Architektur (MVC)

Laut `http://de.wikipedia.org/wiki/Model_View_Controller` heißt MVC im Deutschen »Modell-Präsentation-Steuerung«. Das macht die Sache schon viel einfacher. ;-)

MVC ist eine Struktur zur Software-Entwicklung. Man hat sich einfach darauf geeinigt, dass ein Teil der Software an dieser Stelle und ein anderer Teil der Software immer an einer anderen Stelle ist. Nicht mehr und nicht weniger. Diese Einigung birgt den enormen Vorteil, dass man nach einer gewissen Einarbeitungszeit genau weiß, wo man welche Funktionalität in einem Rails-Projekt suchen bzw. neu einbauen muss.

Model

»Model« steht hier für Datenmodel. Per Default handelt es sich bei Rails-Applikationen um ein ActiveRecord-Datenmodell (siehe Kapitel 4, ActiveRecord).

Alle Models finden Sie im Verzeichnis `app/models/`.

View

Der »View« ist für die Präsentation der Applikation zuständig. Er übernimmt das Rendern der Webseite, einer XML- oder JSON-Datei. Ein View könnte aber auch ein PDF oder einen ASCII-Text rendern. Das hängt ganz von Ihrer Applikation ab.

Alle Views finden Sie im Verzeichnis `app/views/`.

Controller

Nachdem ein Webseitenaufruf in einer Route (siehe Kapitel 6, Routen (routes)) gelandet ist, kommt er von dort in den Controller. Die Route gibt dabei als Ziel eine bestimmte Methode (Action) an. Diese Methode kann dann gewünschte Aufgaben (z. B. einen bestimmten Datensatz suchen und in einer Instanz-Variable abspeichern) erfüllen und lässt danach den gewünschten View rendern.

Alle Controller finden Sie im Verzeichnis `app/controllers/`.

3.8.2 Helper

Eine Helper-Methode nimmt Ihnen wiederholende Arbeiten in einem View ab. Wenn Sie z. B. bei einer Hotel-Bewertung Sterne (*) und nicht eine Zahl von 1 bis 5 anzeigen wollen, so können Sie folgenden Helper in der Datei `app/helpers/application_helper.rb` definieren:

```
module ApplicationHelper

  def render_stars(value)
    output = ''
    if (1..5).include?(value)
      value.times { output += '*'}
    end
    output
  end

end
```

Mit diesem Helper können wir dann in einem View den folgenden Code anwenden:

```
<p>
<b>Bewertung:</b> <%= render_stars(5) %>
</p>
```

Sie können einen Helper auch in der Console ausprobieren:

```
MacBook:webshop xyz$ rails console
Loading development environment (Rails 3.2.5)
1.9.3-p194 :001 > helper.render_stars(5)
 => "*****"
1.9.3-p194 :002 > exit
MacBook:webshop xyz$
```

Es gibt von Rails eine Menge vordefinierter Helper, die wir in den nächsten Kapiteln benutzen werden. Sie können aber auch selber Helper definieren. Dabei können alle Helper aus der Datei `app/helpers/application_helper.rb` in jedem View angewendet werden. Helper, die nur in bestimmten Views verfügbar sein sollen, müssen pro Controller definiert werden. Beim Erstellen eines Controllers wird automatisch in `app/helpers` eine Datei für Helper dieses Controllers erstellt. Diese bietet die Möglichkeit, Helper nur für diesen Controller bzw. für die Views dieses Controllers zu definieren.

Alle Helper finden Sie im Verzeichnis `app/helpers/`.

3.8.3 DRY – Don't repeat yourself

Viele Rails-Programmierer sind große DRY-Fans. DRY heißt nichts anderes, als dass man versuchen sollte, sich wiederholende Programmierlogik in eigene Methoden auszulagern.

Refactoring

Im Zusammenhang von DRY wird oft das Wort Refactoring benutzt. Dabei geht es um funktionierende Applikationen, die weiter optimiert werden. Die Applikation an sich verändert dabei nicht Ihre Oberfläche. Sie wird im Kern aber u. a. durch DRY optimiert.

3.8.4 Convention Over Configuration

»Convention over configuration« (auf Deutsch »Konvention vor Konfiguration« siehe `http://de.wikipedia.org/wiki/Konvention_vor_Konfiguration`) ist ein wichtiger Grundpfeiler einer Rails-Applikation. Es sagt aus, dass der Programmierer sich beim Start eines Projektes nicht erst für bestimmte Features entscheiden und diese per Konfigurationsparameter einstellen muss. Es gibt einen Grund-Konsens und dieser ist per Default eingestellt. Erst wenn man außerhalb dieses Grund-Konsenses arbeiten will, muss man die entsprechenden Parameter verändern.

Copyright