

ralf WIRDEMANN
thomas BAUSTERT



RAPID WEB DEVELOPMENT MIT RUBY ON RAILS

»Ruby on Rails hat die Weichen neu gestellt in puncto Produktivität, Agilität und Einfachheit bei der Entwicklung modernster Webapplikationen. Verpassen Sie nicht den Anschluss – dieses Buch ist Ihr Ticket zu einer bisher ungeahnten Leichtigkeit in der Softwareentwicklung für das neue Web.«

Frank Westphal, Extreme Programmer & Coach

3. Auflage

HANSER



Wirdemann/Baustert

Rapid Web Development mit Ruby on Rails



Bleiben Sie einfach auf dem Laufenden:
www.hanser.de/newsletter

Sofort anmelden und Monat für Monat
die neuesten Infos und Updates erhalten.

Ralf Wirdemann
Thomas Baustert

Rapid Web Development mit Ruby on Rails

3., überarbeitete Auflage

HANSER

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

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht, auch nicht für die Verletzung von Patentrechten und anderen Rechten Dritter, die daraus resultieren könnten. Autoren und Verlag übernehmen deshalb keine Gewähr dafür, dass die beschriebenen Verfahren frei von Schutzrechten Dritter sind.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2008 Carl Hanser Verlag München Wien (www.hanser.de)
Lektorat: Margarete Metzger
Herstellung: Irene Weillhart
Umschlagdesign: Marc Müller-Bremer, Rebranding, München
Umschlagrealisation: MCP • Susanne Kraus GbR, Holzkirchen
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-41498-3

Für meine liebe Freundin Astrid

– Ralf Wirdemann

Für meine Familie und Silvie

– Thomas Baustert

Inhaltsverzeichnis

1	Einleitung	1
1.1	Für wen dieses Buch bestimmt ist	2
1.2	Organisation des Buches	3
1.3	Web-Site zum Buch	5
1.4	Feedback	5
2	Überblick und Installation	7
2.1	Was ist Ruby on Rails?	7
2.2	Bestandteile von Rails	10
2.2.1	Komponenten und Zusammenspiel	10
2.2.2	Action Pack	11
2.2.3	Active Record	11
2.2.4	Action Mailer	12
2.2.5	Ajax	12
2.2.6	Unit Tests	12
2.3	Installation	12
2.3.1	Windows	12
2.3.2	Mac	13
2.3.3	Linux	13
2.3.4	Aktualisierung	13
2.3.5	Datenbank	13
2.3.6	Glückwunsch! Willkommen an Bord!	13
3	Hands-on Rails	15
3.1	Entwicklungsphilosophie	16
3.2	Domain-Modell	16
3.3	OnTrack Product Backlog	17
3.4	Aufsetzen der Infrastruktur	18
3.5	Projekte erfassen, bearbeiten und löschen	20

3.5.1	Modell erzeugen	20
3.5.2	Datenbankmigration	22
3.5.3	Controller	26
3.6	Iterationen hinzufügen	27
3.7	Zwischenstand	34
3.8	Iterationen anzeigen	34
3.9	Iterationen bearbeiten und löschen	36
3.10	Tasks hinzufügen	39
3.11	Tasks anzeigen, bearbeiten und löschen	42
3.12	Struktur in die Seiten bringen	44
3.13	Validierung	45
3.14	Benutzerverwaltung	49
3.15	Login	51
3.16	Tasks zuweisen	54
3.17	Endstand und Ausblick	55
4	Active Record	57
4.1	Active Record-Klassen	57
4.1.1	Mehr über Modellattribute	59
4.1.2	Mehr über Primärschlüssel	60
4.1.3	Zusammengesetzte Primärschlüssel	61
4.1.4	Mehr über Tabellennamen	61
4.2	Active Record direkt verwenden	63
4.3	Die Rails-Konsole	63
4.4	Objekte erzeugen, laden, aktualisieren und löschen	64
4.4.1	Erzeugung	65
4.4.2	Objekte laden	65
4.4.3	Objekte aktualisieren	66
4.4.4	Objekte löschen	68
4.5	Mehr über Finder	68
4.5.1	Suchbedingungen: conditions	68
4.5.2	SQL-Injection vermeiden	70
4.5.3	Ordnung schaffen: order	71
4.5.4	Limitieren: limit	71
4.5.5	Seitenweise: limit und offset	71
4.5.6	Weitere Parameter: joins und include	72
4.6	Dynamische Finder	73

4.7	Kann ich weiterhin SQL verwenden?	74
4.8	Metadaten – Daten über Daten	75
4.9	Assoziationen	76
4.9.1	Grundsätzliches	76
4.9.2	1:1-Beziehungen: <code>has_one</code> – <code>belongs_to</code>	78
4.9.3	1:N-Beziehungen: <code>has_many</code> – <code>belongs_to</code>	84
4.9.4	N:M-Beziehungen: <code>has_and_belongs_to_many</code>	90
4.9.5	Polymorphe Assoziationen: <code>has_many</code> – <code>belongs_to</code>	94
4.9.6	<code>has_many :through</code>	97
4.10	Aggregation	99
4.11	Vererbung	100
4.12	Transaktionen	103
4.13	Von Bäumen und Listen	105
4.13.1	<code>acts_as_list</code>	106
4.13.2	<code>acts_as_tree</code>	108
4.14	Validierung	110
4.14.1	Validierungs-Klassenmethoden	111
4.15	Callbacks	115
4.15.1	Überschreiben von Callback-Methoden	116
4.15.2	Callback-Makros	116
4.15.3	Observer	118
4.16	Konkurrierende Zugriffe und Locking	119
4.16.1	Optimistisches Locking	119
5	Action Controller	121
5.1	Controller-Grundlagen	121
5.1.1	Actions	122
5.1.2	Responses	123
5.2	Datenaustausch	126
5.2.1	Vom Controller zum View	126
5.2.2	Vom View zum Controller	127
5.2.3	Aus der Action in den View und zurück	129
5.3	Redirects	130
5.4	GET vs. POST	131
5.5	Unterschiedliche Response-Formate: <code>respond_to</code>	133
5.6	Zugriff auf Datensätze einschränken	135
5.7	Ausnahme fangen mit <code>rescue_from</code>	136

5.8	Sessions	137
5.8.1	Session-Daten löschen	139
5.9	Der Flash-Speicher	140
5.9.1	Weitere Flash-Methoden	141
5.10	Filter	141
5.10.1	Around-Filter	142
5.10.2	Bedingungen	143
5.10.3	Filterklassen und Inline-Filter	144
5.10.4	Filtervererbung	144
5.10.5	Filterketten	145
5.11	HTTP-Authentifikation	146
5.12	Routing und URL-Generierung	147
5.12.1	Routing	147
5.12.2	Anpassung des Routings	149
5.12.3	Root Route	150
5.12.4	Named Routes	150
5.12.5	URL-Generierung	152
5.13	Layout	153
5.13.1	Automatische Layoutzuweisung	154
5.13.2	Explizite Layoutzuweisung	154
5.13.3	Dynamische Bestimmung des Layouts	155
5.13.4	Action-spezifische Layouts	155
5.14	Datei-Download	156
5.14.1	Die Methode send_data	156
5.14.2	Die Methode send_file	157
5.14.3	Unsichere Dateidownloads verhindern	157
5.15	Datei-Upload	158
5.16	Rails-Konsole für Controller nutzen	158
6	Action View	161
6.1	HTML-Templates	161
6.2	Helper-Module	163
6.3	Action View Helper	165
6.3.1	Formulare	165
6.3.2	Schutz vor Cross-Site Request Forgery	167
6.3.3	Formular-Helper mit Bezug zu Modellen	168
6.3.4	Formular-Helper ohne Bezug zu Modellen	170

6.3.5	HTML-Tags	171
6.3.6	Texte und Zahlen	171
6.3.7	Datum und Zeit	172
6.3.8	Auswahlboxen	173
6.3.9	Verweise und URLs	175
6.3.10	Ressourcen einbinden	177
6.3.11	JavaScript	178
6.3.12	Code speichern und wiederverwenden	178
6.3.13	Debugging	179
6.3.14	HTML-Code filtern	179
6.4	Layouts	180
6.4.1	Layout aus Views beeinflussen	180
6.5	Partial Views	181
6.6	Anzeige Fehlermeldungen	185
6.7	XML-Templates	186
6.8	RJS-Templates	187
6.9	Caching	187
6.9.1	Seiten-Caching	188
6.9.2	Action-Caching	188
6.9.3	Fragment-Caching	190
6.9.4	Was Sie nicht cachen sollten	193
6.9.5	Caching testen	193
6.9.6	Action Cache Plugin	193
7	RESTful Rails	195
7.1	Was ist REST?	196
7.2	Warum REST?	197
7.3	Was ist neu?	197
7.4	Vorbereitung	198
7.5	Ressource Scaffolding	198
7.6	Das Modell	199
7.7	Der Controller	199
7.7.1	REST-URLs	200
7.7.2	REST-Actions verwenden <code>respond_to</code>	201
7.7.3	Accept-Feld im HTTP-Header	202
7.7.4	Formatangabe via URL	203
7.8	REST-URLs in Views	204

7.8.1	New und Edit	205
7.8.2	Path-Methoden in Formularen: Create und Update	206
7.8.3	Destroy	208
7.9	URL-Methoden im Controller	209
7.10	REST-Routing	210
7.10.1	Konventionen	210
7.10.2	Customizing	210
7.11	Verschachtelte Ressourcen	211
7.11.1	Anpassung des Controllers	213
7.11.2	Neue Helper für verschachtelte Ressourcen	214
7.11.3	Zufügen neuer Iterationen	216
7.11.4	Bearbeiten existierender Iterationen	218
7.12	Eigene Actions	219
7.12.1	Sind wir noch DRY?	221
7.13	Eigene Formate	221
7.14	RESTful AJAX	222
7.15	Testen	224
7.16	RESTful Clients: ActiveResource	225
7.17	Abschließend	226
8	Internationalisierung	227
8.1	Internationalisierung oder Lokalisierung?	228
8.2	Lokalisierung und Codes	228
8.2.1	Language Tag	228
8.2.2	Locale	228
8.3	Checkliste	229
8.4	Internationalisierung vorbereiten	230
8.5	Ruby Gettext-Package	231
8.5.1	Installation	232
8.5.2	Texte übersetzen	232
8.5.3	Die Schritte im Überblick	237
8.5.4	Anwendung und Gettext zusammenbringen	237
8.5.5	Lokalisierte Templates	238
8.5.6	Controller	238
8.5.7	Modelle	239
8.5.8	Dynamische Texte und Pluralisierung	241
8.5.9	Datum, Zahlen und Währungen	243

8.5.10	Wochen- und Monatsnamen	245
8.5.11	Sortierung	247
8.5.12	Zusammenfassung	248
9	Action Mailer	249
9.1	E-Mail-Versand	249
9.1.1	E-Mail erstellen	249
9.1.2	E-Mail-Objekt erzeugen	253
9.1.3	E-Mail versenden	254
9.1.4	Testen	256
9.1.5	Tipp zum zuverlässigen E-Mail-Versand	257
9.1.6	Multipart E-Mails	258
9.2	E-Mail-Empfang	260
9.2.1	E-Mail empfangen	261
9.2.2	Empfang testen	262
10	Ajax	263
10.1	Ajax und Rails	265
10.2	Hello, Ajax World	266
10.3	Ajax-Formulare	267
10.3.1	Task-Schnellerfassung	267
10.3.2	Ajax-Formulare und JavaScript	270
10.4	Feldbeobachter	270
10.4.1	Beispiel: Tasks filtern	272
10.5	Callback-Methoden	274
10.6	Drag and Drop	275
10.6.1	Einkaufen per Drag and Drop	275
10.6.2	Sortieren per Drag and Drop	278
10.7	Autocompletion	281
10.8	RJS Templates	282
10.8.1	Hinzufügen von Tasks visuell optimieren	283
10.8.2	Weitere Techniken	286
10.8.3	Debugging und Testen	289
10.9	Zusammenfassung und Ausblick	290
11	Produktion	291
11.1	Umgebungen in Rails	291
11.1.1	Umgebung definieren	291

11.1.2	Entwicklung	292
11.1.3	Test	292
11.1.4	Produktion	292
11.1.5	Umgebung hinzufügen	293
11.2	Webserver und Datenbank	294
11.2.1	Datenbank	294
11.2.2	Webserver Cluster	294
11.2.3	Apache und Mongrel	295
11.2.4	Alternativen	298
11.3	Skalierung	299
11.4	Session-Speicher	300
11.5	Tipps zur Performanz	301
11.6	Sicherheit	302
11.7	Auslieferung	305
11.8	Wartung	305
11.9	Überwachung	306
12	Deployment mit Capistrano (und Subversion)	309
12.1	Quickstart: Capistrano in 10 Minuten	310
12.1.1	Voraussetzungen	310
12.1.2	Installation von Capistrano	312
12.1.3	Anwendung Capistrano-ready machen	312
12.1.4	Konfiguration	313
12.1.5	Setup des entfernten Verzeichnisses	315
12.1.6	Erstes Deployment	315
12.1.7	Fallstricke	318
12.1.8	Nachfolgende Deployments	319
12.2	Datenbanksetup und Migration	319
12.3	Rollback eines Release	320
12.3.1	Rollback mit Datenbankmigration	320
12.4	Tasks	320
12.4.1	Ausführen von Tasks	321
12.4.2	Mehr über run und sudo	323
12.4.3	Weitere Task-Helper: delete, put und render	324
12.4.4	Transaktionen und Rollbacks	324
12.4.5	Überschreiben von Standardtasks	325
12.4.6	Tasks erweitern: Before- und After-Tasks	325

12.5 Variablen	326
12.6 FastCGI-Utilities	327
12.6.1 Spawner	327
12.6.2 Reaper	328
12.7 Gemeinsame Dateien – das Shared-Verzeichnis	328
12.7.1 Eine „persistente“ Datenbank-Konfiguration	328
12.8 Capistrano-Referenz	330
12.9 Capistrano-Konfiguration: deploy.rb	331
13 Testgetriebene Entwicklung mit Ruby und Test::Unit	333
13.1 Unit Tests – eine Definition	333
13.2 Ein Beispiel	334
13.3 Warum testen?	335
13.4 Test::Unit	336
13.4.1 Strukturierung von Unit Tests	337
13.4.2 Wohin mit den Tests?	339
13.4.3 Ausführen der Tests	339
13.4.4 Unabhängigkeit von Tests	340
13.5 Testgetriebene Softwareentwicklung	340
13.5.1 TODO-Listen	342
13.5.2 Beispiel	342
13.6 Retrospektive	344
14 Testgetriebene Entwicklung mit Ruby on Rails	345
14.1 Generierte Testklassen	345
14.2 Testdatenbank	346
14.3 Testausführung mit Rake	347
14.4 Eine Programmierepisode	347
14.4.1 Entwicklung einer Modellklasse	348
14.4.2 Entwicklung des Controllers	352
14.4.3 Programmierung des Views	353
14.4.4 Geänderte Anforderungen	355
14.4.5 Retrospektive	361
14.5 Unit Tests – Testen von Modellklassen	362
14.5.1 Struktur und Elemente von Modelltests	362
14.5.2 Testmethoden	363
14.5.3 Testdaten – Fixtures	364
14.5.4 Transaktionale Fixtures	367

14.5.5	Testrezepte für Modelle	367
14.6	Funktionale Tests	373
14.6.1	Testmethoden	373
14.6.2	Kontrollfluss-Assertions	374
14.6.3	Routing-Assertions	376
14.6.4	Datencontainer-Assertions	377
14.6.5	Template-Assertions	377
14.6.6	Testrezepte für Controller und Views	379
14.7	Integrationstests	383
14.7.1	Test-DSLs	385
14.7.2	Sessions	386
14.8	Testen von E-Mails	387
14.9	Mock-Objekte	387
14.10	Zusammenfassung	388
14.11	Assertions – Übersicht	389
Anhang		391
Literaturverzeichnis		399
Stichwortverzeichnis		401

Vorwort

Als wir im Herbst 2004 über ein Blog-Posting von David Heinemeier Hansson auf Ruby on Rails gestoßen sind, konnten wir nicht absehen, welche Bedeutung Rails für die Web-Entwicklung im Allgemeinen und für unsere eigene Softwareentwicklung im Speziellen haben würde.

Nach vielen Jahren Java- und insbesondere J2EE-Entwicklung waren wir zunächst einmal überrascht, wie einfach Softwareentwicklung sein kann. Eigentlich waren wir so überrascht, dass wir anfangs nicht glauben konnten, dass sich Rails wirklich für die Entwicklung größerer Web-Anwendungen eignen würde.

Die Monate November und Dezember des Jahres 2004 verbrachten wir mit der Entwicklung kleinerer Rails-Applikationen, um zunächst die Technologie kennen zu lernen. Diese Erfahrung hat unsere anfängliche Skepsis deutlich gemindert, zumal wir mehr und mehr feststellen konnten, dass Softwareentwicklung mit Rails nicht nur einfach ist, sondern darüber hinaus auch zu sauber entworfenen Systemen führt, die sich durch ihre Wartbarkeit und damit Langlebigkeit auszeichnen.

Seit Januar 2005 entwickeln wir Rails-Applikationen im kommerziellen Umfeld. Was wir seitdem täglich neu erfahren, ist eine bisher nicht gekannte Produktivität und eine neue „Leichtigkeit der Softwareentwicklung“, die wir Ihnen mit diesem Buch nahebringen wollen.

Hamburg, im November 2005

Ralf Wirdemann und Thomas Baustert

Vorwort zur 2. Auflage

Fast zeitgleich mit Abschluss der Arbeiten an der 2. Auflage unseres Buches hat Rails am 25. Juli 2006 seinen 2. Geburtstag gefeiert. Nach nunmehr zwei Jahren Rails ist eines sicher: Wenn die Welt ein neues Web-Framework brauchte, dann dieses.

Konferenzen, Zeitschriftenartikel, Bücher und eine ständig zunehmende Anzahl an Rails-Projekten zeigen dies. Dabei sind es nicht nur kleine Internetagenturen, die ihre Entwicklung auf Rails umstellen, sondern auch große Firmen, die vorhandene JEE-Lösungen portieren oder neue Projekte auf Basis von Rails starten.

Rails- und Ruby-Konferenzen sind innerhalb kürzester Zeit ausverkauft. Während die seit Jahren in den USA stattfindende Ruby-Konferenz noch bis zum Jahr 2004 eher ein Nischendasein fristete, war die im Oktober 2006 stattfindende Ruby-Konferenz in Denver innerhalb von vier Stunden nach Öffnung der Registrierungs-Website ausverkauft.

Die Popularität von Rails ist längst aus den USA zu uns nach Europa und in andere Teile der Welt übergeschwappt. Die deutsche Rails Community wächst ständig. Lokale Usergruppen organisieren regelmäßige Treffen. Die erste deutsche Rails-Konferenz steht in den Startlöchern. Die Anzahl der Rails-Projekte in Deutschland nimmt kontinuierlich zu.

Rails hat der Sprache Ruby zu neuem Ruhm verholfen. Dies zeigen nicht zuletzt die von O'Reilly veröffentlichten Verkaufszahlen: Im Jahr 2005 wurden 1552% mehr Ruby-Bücher verkauft, als im Vergleichszeitraum des Vorjahres. Viele Entwickler haben die Eleganz und Ausdrucksstärke von Ruby kennen und schätzen gelernt. Wir sind vielen skeptischen Entwicklern begegnet, wenn es um den Umstieg von Java auf Ruby ging. Wir sind bisher jedoch keinem Entwickler begegnet, der nach erfolgreichem Umstieg zurück in die Java-Welt wollte.

Das Rails-Framework wurde in den letzten zwei Jahren kontinuierlich weiterentwickelt und verbessert. Rails bleibt dabei trotzdem schlank und einfach. Neue Features werden nur dann ins Framework aufgenommen, wenn sie sich in der Praxis bewährt haben und allgemeinen Nutzen versprechen. Ein Beispiel hierfür sind die seit Rails 1.1 verfügbaren Integrationstests, die von 37signals im Rahmen ihrer Campfire-Software entwickelt wurden und erst nach ihrem erfolgreichen Einsatz Einzug in das Rails-Framework hielten. Ein anderes Beispiel für eine einfache, dafür aber umso wirkungsvollere Verbesserung sind Active Record-Migrations, die

die bis Rails 1.0 verwendeten SQL-Skripte ersetzen und die inkrementelle Pflege von Datenbankschemata ermöglichen.

Trotz anhaltender Euphorie gibt es weiterhin viel zu tun. Z.B. zeichnet sich auch nach zwei Jahren kein eindeutiger Favorit am Internationalisierungshimmel ab. Diese Erkenntnis war für uns Anlass genug, das Internationalisierungskapitel der ersten Auflage vollständig neu zu schreiben und die aktuell verfügbaren und praxiserprobten Lösungen vorzustellen.

Aber auch für uns persönlich hat sich in den letzten zwei Jahren vieles geändert. Geblieben ist die Begeisterung für Rails als Web-Framework und Ruby als Programmiersprache. Hinzugekommen sind eine Menge neuer Erfahrungen, viele Leute, die wir im letzten Jahr kennen gelernt haben und die inzwischen selbst von PHP oder Java auf Rails umgestiegen sind.

Unser Buch versteht sich weiterhin als Ein- und Aufsteigerbuch für die Ruby on Rails-Entwicklung. Wir haben versucht, die 2. Auflage unseres Buches entscheidend zu verbessern. Neben vielen Korrekturen und Anmerkungen unserer Leser enthält die Neuauflage alle wesentlichen Änderungen von Rails 1.1¹:

- Active Record-Migrations
- neue Active Record-Assoziationen
- RJS-Templates
- Formulare mit `form_for()`
- Unterstützung unterschiedlicher Clients mit `response.to()`
- Integrationstests
- Nutzung von Apache und Mongrel

Darüber hinaus enthält die zweite Auflage ein ausführliches Kapitel zum Thema Deployment mit Capistrano, dem Standardwerkzeug für die automatisierte Auslieferung und Verteilung von Rails-Applikationen.

Wir wünschen Ihnen viel Spaß und Freude beim Durcharbeiten dieses Buches und vor allem Produktivität und Erfolg für Ihr nächstes Ruby on Rails-Projekt.

Hamburg, im September 2006

Ralf Wirdemann und Thomas Baustert

¹ bzw. von Rails 1.0, sofern sie es nicht in die 1. Auflage geschafft haben.

Vorwort zur 3. Auflage

Mit Erscheinen der dritten Auflage unseres Buches ist Ruby on Rails mehr als 3 Jahre alt und hat die Versionsnummer 2 erreicht. Rails 2 markiert einen weiteren Meilenstein in der Entwicklung des Frameworks. Neben vielen kleineren Neuerungen dürfte das Thema REST die wohl einschneidendste Neuerungen dieses Major-Releases sein.

REST ist zwar schon seit Rails 1.2 fester Bestandteil des Frameworks, seit Rails 2 aber zum bevorzugtem Paradigma für die Entwicklung von Web-Applikationen mit Rails geworden. Entwickler müssen umdenken und dabei eine neue Sichtweise aufs Web entwickeln: Web-Applikationen sind nunmehr keine Ansammlung dynamischer Webseiten mehr, sondern vielmehr eine Menge miteinander verbundener Ressourcen, deren HTML-Repräsentation im Browser nur eine von vielen möglichen Repräsentationsvarianten ist.

Eine logische Konsequenz aus der Verwendung von Ressourcen ist Erweiterung von Rails um Multiview-Fähigkeit: Controller erkennen das vom Client gewünschte Ressourcen-Format und reagieren darauf durch Auslieferung eines bestimmten Templates, z.B. *index.html.erb* für normale Browser-Requests, oder *index.iphone.erb* für Requests von mobilen Internet-Geräten.

Weitere Änderungen gab es in den Frameworks Action Pack und Active Record. Zum Beispiel können Ressourcen zukünftig direkt an Helper in Controllern und Views übergeben, was den Source-Code noch einmal schlanker und lesbarer macht. Einige Beispiele: *redirect_to(@person)*, *link_to(@person.name, @person)* oder *form_for(@person)*. Im Bereich Active Record werden Migrationen einfacher und Fixtures übersichtlicher.

Rails hat aber nicht nur zugenommen, sondern auch abgespeckt: Das Subframework Action Web Service gibt es nicht mehr. Durch die konsequente Umstellung auf REST hat ein neues Web-Service Paradigma Einzug in die Rails-Welt gehalten. RESTful entwickelte Anwendungen benötigen kein spezielles Web-Service Interface mehr, da die Anwendung von Haus aus eine REST-Schnittstelle zur Verfügung stellt, das nicht nur von Browsern, sondern von jedem REST sprechenden Client genutzt werden kann. Die Anwendung wird so zur API.

Sie sehen schon, Rails 2 ist vollgepackt mit vielen großen, aber auch kleinen wichtigen Änderungen und Verbesserungen. Wir haben unser Buch vollständig überar-

beitet und um die Rails 2 spezifischen Änderungen erweitert und die auf älteren Rails-Versionen basierenden Beispiele angepasst.

Wir bedanken uns bei unseren Kollegen und Reviewern Sascha Teske und Michael Voigt für ihre Korrekturarbeit und das Testen unserer Beispiele.

Hamburg, im Mai 2008

Ralf Wirdemann und Thomas Baustert

Danksagung

An der Entstehung dieses Buches waren viele Personen beteiligt, bei denen wir uns bedanken möchten. Unser Dank gilt zunächst einmal unseren Familien und Freunden für die über Monate andauernde Unterstützung. Bei Frau Metzger und Frau Weilhart vom Carl Hanser Verlag bedanken wir uns für die außergewöhnlich gute Betreuung und das uns entgegengebrachte Vertrauen.

Darüber hinaus möchten wir uns bei der Firma Carl Schröter und insbesondere deren Mitarbeiter Klaus Scheler bedanken, der uns in seiner Rolle als Kunde in einem sehr agilen Rails-Projekt hervorragend unterstützt hat. Nur durch seine immer neuen und nie versiegenden Anforderungen war es uns möglich, Rails in der notwendigen Tiefe kennen zu lernen, um aufbauend auf diesem Wissen ein Buch darüber zu schreiben.

Unseren Reviewern der 1. Auflage

Ganz besonders möchten wir uns bei unseren Reviewern bedanken, die uns über Wochen mit Kritik und Ratschlägen zur Seite gestanden haben:

- Astrid Ritscher
- Dr. Richard Oates
- Frank Westphal
- Konrad Riedel
- Michael Schürig
- Tammo Freese
- Torsten Lückow

Eine besondere Rolle nimmt dabei Astrid Ritscher ein, die mit ihrer Kreativität und ihren Ideen federführend das Layout der Umschlaginnenseiten dieses Buches gestaltet hat.

Unseren Lesern und Reviewern der 2. Auflage

Wir wollen uns mit der 2. Auflage unseres Buches bei allen Menschen bedanken, die unser Buch gelesen und uns so viel wertvolles Feedback, Ideen und Verbesserungsvorschläge geliefert haben. Zu nennen sind hier insbesondere: Thorsten Brückner, Markus Fink, Tammo Freese, Paul Fühling, Johannes Held, Marco Kratzenberg, Vico Klump, Peter-Hinrich Krogmann, Ingo Paulsen, Reiner Pittinger, Axel Rose, Dirk V. Schesmer und Stefan Schuster.

Darüber hinaus möchten wir unser „offizielles“ Review-Team der 2. Auflage nennen und uns für die intensive Arbeit und die guten Anregungen bedanken. Durch euch ist unser Buch viel besser geworden.

Ingo Paulsen

Ein riesengroßes Dankeschön geht an Ingo Paulsen. Wir haben Ingo durch sein überaus qualifiziertes Feedback zur 1. Auflage kennen gelernt und ihn daraufhin gefragt, ob er die 2. Auflage nicht vor ihrem offiziellen Erscheinen lesen möchte. Ingo hat viel mehr für uns getan, als wir uns erhofft hatten. Neben seinem hervorragenden Feedback hat uns sein weit überdurchschnittliches Engagement jedes Wochenende aufs neue motiviert.

Bernd Schmeil und Timo Hentschel von AutoScout24

Bernd und Ingo waren Teilnehmer eines unserer Rails-Workshops in München. Ohne die beiden gäbe es keine Hinweise zur Verwendung von Subversion im Capistrano-Kapitel.

Uwe Petschke von ObjectFab

Uwe ist ein ehemaliger Kollege und war Teilnehmer unseres ersten Rails-Workshops in Dresden.

Mathias Meyer

Mathias ist ein Rails-Pionier aus Berlin. Wir wünschen Mathias, dass er demnächst die Zeit findet, um seine (PHP-basierte) Beatsteaks-Site endlich auf Rails umzustellen.

Johannes Held

Johannes hat uns kontinuierlich mit Anmerkungen und Verbesserungsvorschlägen sowohl zu unserer Erstauflage als auch zu den überarbeiteten Teilen der neuen Auflage versorgt.

Thorsten Brückner

Thorsten ist Berater und Softwareentwickler und hat zur Klarstellung einiger Aspekte beigetragen.

Andreas Bürk und Florian Görsdorf

Andreas und Florian sind Mitglieder unseres Wunderloop-Teams und haben mit ihrem Last-Minute-Review letzte Ungereimtheiten unseres Hands-on Tutorials beseitigt.

Astrid Ritscher

Astrid war immer da und hat alle neuen oder überarbeiteten Abschnitte als Erste an meinem Bildschirm gelesen und direkt korrigiert.

Kapitel 1

Einleitung

Herzlich willkommen zu *Rapid Web Development mit Ruby on Rails*. Das vorliegende Buch ist eine umfassende Einführung in die Entwicklung von Datenbank-basierten Web-Anwendungen mit Ruby on Rails (kurz Rails). Die Inhalte dieses Buches beruhen im Wesentlichen auf den praktischen Erfahrungen einer von uns entwickelten B2B-Anwendung. Wir wurden dabei viele Male positiv von Rails überrascht, hatten aber auch einige Klippen zu umschiffen, wie z.B. fehlende Bibliotheken oder unzureichende Internationalisierungs-Konzepte.

Hauptziel dieses Buches ist es, Ihnen die Entwicklung von Web-Anwendungen mit Rails praxisorientiert, zielstrebig und mit raschen Erfolgserlebnissen nahezubringen, ohne dass Sie dabei die von uns bereits umschifften Klippen erneut umfahren müssen. Nach der Lektüre werden Sie in der Lage sein, Datenbank-basierte und geschäftskritische Web-Anwendungen eigenständig und mit einer bisher nicht gekannten Produktivität zu entwickeln und diese Anwendungen in Produktion zu bringen.

Neben der praxisorientierten Einführung in Rails verfolgen wir mit diesem Buch auch folgende Ziele:

- die Vermittlung der für das Verständnis und den praktischen Einsatz von Rails notwendigen Ruby-Kenntnisse;
- die Internationalisierung und Lokalisierung von Rails-Anwendungen;
- die Einführung in die testgetriebene Web-Entwicklung mit Ruby on Rails.

Rails konfrontiert den Software-Entwickler gleichermaßen mit zwei neuen Technologien: der Programmiersprache Ruby und dem Web-Framework Rails. Wir haben versucht, dieses Buch so zu gestalten, dass es auch für Leser ohne Ruby-Kenntnisse verständlich ist, dabei aber die für Rails benötigten Ruby-Grundkenntnisse nebenbei und quasi parallel zu Rails mit einführt. Unser ursprünglicher Plan war es, die für die Rails-Entwicklung benötigten Ruby-Kenntnisse in Form eines eigenständigen und abgeschlossenen Ruby-Kapitels zu vermitteln. Während des Schreibens und

insbesondere durch das Feedback unserer Reviewer mussten wir jedoch feststellen, dass ein Ruby-Kapitel vom eigentlichen Kern dieses Buches ablenkt, insbesondere dann, wenn es am Anfang des Buches steht.

Deshalb haben wir uns entschieden, auf das Ruby-Kapitel¹ zu verzichten und gleich zu Beginn in Form eines Hands-On-Tutorials in die Rails-Entwicklung einzusteigen. Im Laufe des Tutorials tauchen dabei immer wieder Ruby-spezifische Konstrukte und Begriffe auf, die wir an der Stelle ihres ersten Vorkommens in einem optisch hervorgehobenen Kasten beschreiben. Die auf diese Art vermittelten Ruby-Kenntnisse sind völlig ausreichend, um die Inhalte dieses Buchs zu verstehen und mit der praktischen Rails-Entwicklung zu beginnen.

Da Rails keinen Standard für die Internationalisierung von Web-Anwendungen enthält, ist es uns gerade im Hinblick auf ein deutschsprachiges Buch besonders wichtig, Ihnen unsere Erfahrungen mit der Internationalisierung von Rails-Anwendungen zu beschreiben. Aus diesem Grunde zeigen wir Ihnen in einem eigenen Kapitel, wie Sie internationalisierte Rails-Anwendungen entwickeln und die Anwendung damit für die Nutzung in verschiedenen Sprachen vorbereiten.

Als Anhänger der testgetriebenen Softwareentwicklung waren wir von Anfang an von den Möglichkeiten begeistert, die Rails hinsichtlich der Entwicklung von automatisierten Unit Tests bietet. Seit Beginn unseres ersten Rails-Projekts schreiben wir für nahezu jeden Aspekt unserer Anwendungen einen Unit Test – und das in der Regel vor der Entwicklung der eigentlichen Funktionalität. Dieses Vorgehen hat sich sehr schnell automatisiert und ist mittlerweile aus unserem Entwicklungsalltag nicht mehr wegzudenken. Deshalb ist es uns ein weiteres wichtiges Anliegen, Ihnen mit diesem Buch eine grundlegende Einführung in die testgetriebene Web-Entwicklung mit Ruby on Rails zu geben.

Wir wünschen Ihnen viel Spaß bei der Lektüre und eine produktive Zeit bei der Entwicklung Ihrer nächsten und weiteren Rails-Anwendungen.

1.1 Für wen dieses Buch bestimmt ist

Wir haben unser Buch für Web-Entwickler geschrieben, die Freude an der Entwicklung sauber entworfener Web-Anwendungen haben und dies mit einer zuvor nicht gekannten Produktivität betreiben.

Wir setzen Grundkenntnisse in der objektorientierten Programmierung sowie ein Verständnis des Aufbaus und der Funktionsweise dynamischer Web-Applikationen voraus.

Auch für Projektleiter und Manager?

Entwickler sind an spannenden und pragmatischen Technologien interessiert, mit denen sie ihre Ansprüche an Software und deren Entwicklung in die Praxis um-

¹ Das im Rahmen dieses Buches entstandene Ruby-Tutorial steht als kostenloses PDF-Dokument unter www.rapidwebdevelopment.de zum Download bereit.

setzen können. Projektleiter und Manager interessieren sich neben der eigentlichen Technologie für deren Kosten.

Auf den ersten Blick ist Rails einfach nur cool und spannend. Auf den zweiten Blick wird jedoch sehr schnell klar, dass Rails auch den Bedürfnissen der zweiten Gruppe gerecht wird: Rails und eingesetzte Technologien wie Apache oder Linux sind Open Source, und noch viel wichtiger ist die enorme und bisher nicht gekannte Produktivität der Entwicklung mit Rails.

Mit Rails entwickeln Sie sauber entworfene, umfangreich getestete und somit langlebige Web-Anwendungen und sorgen so für einen schnellen Return on Investment (ROI) Ihrer Kunden. Deshalb empfehlen wir unser Buch auch Projektleitern und Managern, die sich mit dem Wechsel auf Rails einen entscheidenden Wettbewerbsvorteil verschaffen wollen.

1.2 Organisation des Buches

Neben der Einleitung, die Sie gerade lesen, besteht unser Buch aus den folgenden Kapiteln und einem Anhang:

■ Kapitel 2: Überblick und Installation

Dieses Kapitel führt in Rails, dessen Komponenten und die Rails zugrunde liegenden Prinzipien ein und gibt einige Hinweise zur Installation von Rails.

■ Kapitel 3: Hands-on Rails

Hier demonstrieren wir die praktische Web-Entwicklung mit Rails an Hand einer vollständigen Beispielanwendung. Das Kapitel stellt Schritt für Schritt die einzelnen Rails-Komponenten vor und ist so geschrieben, dass Sie das entwickelte Beispiel direkt auf Ihrem Rechner nachprogrammieren können und sollen.

■ Kapitel 4: Active Record

Das Kapitel beschreibt das Active Record Framework, eines der drei Sub-Frameworks von Rails. Angefangen von der Abbildung eines Domain-Objekts auf eine Datenbanktabelle bis hin zur Modellierung komplexer Assoziationen finden Sie hier alles, was Sie für den Einsatz von Active Record brauchen.

■ Kapitel 5: Action Controller

Hier wird das Rails-Controller-Framework beschrieben. Controller sind Teil des Sub-Frameworks Action Pack. Controller steuern den Kontrollfluss einer Rails-Anwendung, indem sie HTTP-Requests entgegennehmen, verarbeiten und als Ergebnis eine HTML-Seite an den Client liefern.

■ Kapitel 6: Action View

Das Kapitel schließt an Kapitel 5 an und beschreibt den zweiten Teil des Sub-Frameworks Action Pack: Rails Views sind HTML-Views mit eingebettetem Ruby-Code. Wir zeigen, wie Sie einfache Ruby-Befehle in eine HTML-Seite einbetten, beschreiben einige Helper-Methoden und gehen auf das Thema Caching ein.

■ Kapitel 7: RESTful Rails

Spätestens seit Version 2.0 steht Rails ganz im Zeichen von REST. Zentrale Prinzipien von REST sind so genannte Ressourcen, die mittels Standard-HTTP-Methoden angefordert und manipuliert werden. Dieses Kapitel führt in die Grundlagen von REST ein und beschreibt darauf aufbauend die Entwicklung von REST-basierten Rails-Anwendungen.

■ Kapitel 8: Internationalisierung

Diesem Thema kommt gerade in einem deutschsprachigen Rails-Buch eine besondere Bedeutung zu und wir beschreiben in diesem Kapitel die Internationalisierung mit Hilfe der *Gettext*-Bibliothek.

■ Kapitel 9: Action Mailer

E-Mail-Versand und -Empfang sind typische Funktionen vieler Web-Anwendungen. Beide Funktionen lassen sich mit Rails einfach und pragmatisch umsetzen. Wir demonstrieren dies anhand von Beispielen.

■ Kapitel 10: Ajax

Dieses Kapitel führt in die Ajax-Unterstützung von Rails ein und zeigt Beispiele für die Entwicklung „reicher“ und hoch interaktiver Web-Anwendungen.

■ Kapitel 11: Produktion

Hier geben wir Informationen und Tipps für die Inbetriebnahme von Rails-Anwendungen in Produktion, deren Wartung und Überwachung und einige Hinweise zur Performanz.

■ Kapitel 12: Deployment mit Capistrano (und Subversion)

Capistrano hat sich als De-facto-Standard für die automatisierte und wiederholbare Auslieferung und Verteilung von Rails-Anwendungen etabliert. Dieses Kapitel beschreibt in einem Quickstart-Tutorial, wie Sie Ihre Anwendung auf die Nutzung von Capistrano vorbereiten und deployen. Darauf aufbauend liefert das Kapitel detaillierte Informationen zur Konfiguration und Erweiterung sowie zum effizienten Einsatz von Capistrano.

■ Kapitel 13: Testgetriebene Entwicklung mit Ruby und Test::Unit

Rails wurde speziell im Hinblick auf gute Testbarkeit entwickelt. Dieses Kapitel führt in die Grundlagen der Entwicklung von Unit Tests in Ruby ein und liefert darüber hinaus eine Einführung in das Konzept der testgetriebenen Programmierung.

■ Kapitel 14: Testgetriebene Entwicklung mit Ruby on Rails

Aufbauend auf Kapitel 13 beschreibt dieses Kapitel die Umsetzung der Konzepte der testgetriebenen Programmierung im Rahmen der Rails-Entwicklung. Hier lernen Sie, wie sich nahezu jeder Aspekt einer Rails-Anwendung durch einen Unit Test absichern lässt. Test-Rezepte mit Lösungsbeschreibungen für typische und wiederkehrende Testsituationen runden das Kapitel ab.

■ Anhang

Im Anhang finden Sie Informationen zu Konfigurations-Parametern und Einstellungen der verschiedenen Rails-Komponenten.

1.3 Web-Site zum Buch

Auf der begleitenden Web-Site zum Buch *www.rapidwebdevelopment.de* finden Sie die Errata, Quellcodes und Informationen rund um das Buch.

Außerdem können Sie von dieser Seite das im Rahmen dieses Buches entstandene Ruby-Kapitel als PDF-Dokument herunterladen. Das Kapitel enthält eine allgemeine Einführung in die Sprache Ruby und ihre zugrunde liegenden Konzepte.

1.4 Feedback

Wie freuen uns über Feedback jeglicher Art. Teilen Sie uns Ihre Hinweise, Korrekturen oder sonstigen Anmerkungen per E-Mail unter *ror-buch@b-simple.de* mit. Vielen Dank!

Kapitel 2

Überblick und Installation

Dieses Kapitel liefert einen Einstieg in die Arbeit mit Ruby und Rails. Wir beginnen mit einem kurzen Überblick über Rails und seine Komponenten. Im Anschluss folgen Hinweise zur Installation auf verschiedenen Betriebssystemen. Das Kapitel schafft somit die Grundlage für das Folgekapitel 3, in dem wir Rails anhand eines Beispiels praktisch kennenlernen.

2.1 Was ist Ruby on Rails?

Ruby on Rails¹ oder kurz *Rails* ist ein Web- und Persistenz-Framework für die Programmiersprache Ruby. Im Folgenden sind die Kernpunkte von Rails aufgeführt:

■ Model View Controller

Ruby on Rails basiert auf einer sauberen MVC-Architektur². Es stellt für jede Komponente im MVC-Muster eine unterstützende Komponente bereit. Domain-Objekte, so genannte Modelle, werden mit Hilfe des Frameworks *Active Record* erstellt. Für Ihre Controller und Views stehen *Action Controller* und *Action View* bereit. Die Trennung der Schichten führt zu einer klaren Trennung der Verantwortlichkeiten und zu einer Verringerung von Abhängigkeiten im Code. Dies ist für eine langfristig wartbare Anwendung unabdingbar.

■ Konvention über Konfiguration

Rails ist per Default so ausgelegt, dass Ihre Anwendung ohne umfangreiche Konfiguration auskommt. Rails setzt hier u.a. auf Namenskonventionen. So erhält z.B. eine Datenbanktabelle den Pluralnamen des zugehörigen Domain-Objekts, oder der Name einer Methode, die einen Request verarbeitet, wird aus der URL des eingehenden HTTP-Requests ermittelt. In beiden Fällen kann die Zuordnung ohne Konfiguration erfolgen.

¹ Auf Deutsch: Ruby auf Schienen

² Zu MVC siehe auch <http://de.wikipedia.org/wiki/MVC>

Bei Bedarf besteht die Möglichkeit, mit wenig Aufwand vom Default-Verhalten abzuweichen. Rails bietet also eine Reihe auf praktischen Erfahrungen basierender Defaulteinstellungen, lässt Ihnen aber die Freiheit, diese nach Ihren Wünschen zu ändern.

■ DRY-Prinzip

Die Abkürzung DRY steht für *Don't Repeat Yourself* und wurde von Dave Thomas und Andy Hunt in [4] geprägt. Das DRY-Prinzip besagt, dass Wissen nur eine einzige, eindeutige Repräsentation in einem System hat. Weder Daten noch Funktionalität sollten redundant vorkommen, da andernfalls der Wartungsaufwand beträchtlich erhöht wird.

Rails setzt das DRY-Prinzip konsequent in allen Bereichen um. Dazu zählt z.B., dass Sie für eine Datenbanktabelle weder korrespondierende Attribute noch Getter- und Setter-Methoden in Ihrem Domain-Objekt definieren müssen. Diese Redundanz entfällt, weil Rails entsprechende Attribute und Methoden automatisch erzeugt.

■ Extrahiert

Das Framework wurde aus einer bestehenden Anwendung extrahiert.³ Dies ist für die Handhabbarkeit eines jeden Frameworks zwingend notwendig. Nur so stellt man sicher, dass das Framework den spezifischen Anforderungen der Anwendungen genügt und den Anwender optimal in seiner Entwicklung unterstützt. Im Gegensatz zu Frameworks und Spezifikationen, die auf der grünen Wiese entstehen oder in Gremien erarbeitet werden, hat Rails seine Praxistauglichkeit bereits bewiesen.

■ Ruby

Rails basiert auf der Sprache Ruby. Diese zeichnet sich insbesondere durch ihre verständliche Syntax und erwartungskonforme Semantik aus. Programme werden mit wenig Code geschrieben und drücken dennoch viel aus. Durch die Klarheit des Quellcodes ist dieser auch Monate später noch zu lesen und zu verstehen. Durch die dynamische Typisierung entfällt während der Entwicklung die Zeit für Übersetzung und Deployment. Ein unmittelbares Feedback jeder Änderung ist das Ergebnis.

Ruby unterstützt die einfache Anpassung des Frameworks an eigene Bedürfnisse, falls Rails dies nicht direkt ermöglicht (z.B. per Konfiguration). Ruby erlaubt die nachträgliche Erweiterung von bestehenden Klassen, wodurch Sie gezielt entsprechende Punkte im Code-Verhalten ändern können.

■ Unit Tests

Von Beginn an wurde auf die Testbarkeit von Rails-Anwendungen Wert gelegt. Unit Tests sind in allen MVC-Ebenen leicht möglich. Modelle und Controller testen Sie durch einfache Aufrufe der entsprechenden Methoden. Für die Views kann von der Prüfung des HTTP Return-Werts bis hin zu beliebig tief verschachtelten HTML Tags alles geprüft werden. Mit Rails erstellen Sie sauber getestete Anwendungen.

³ Basecamp, <http://www.basecamphq.com/>

■ Scaffolding

Eine einfache und sofort lauffähige Version Ihrer Anwendung erhalten Sie über das so genannte *Scaffolding* (engl.: Gerüstbau). Dabei werden das Modell, der Controller und einige Views generiert, die zusammen die Erzeugung, Anzeige, Bearbeitung und Speicherung von Modellen ermöglichen. Die Anwendung kann anschließend sukzessiv um individuelle Funktionalität erweitert werden und bleibt dabei zu jedem Zeitpunkt voll funktionsfähig.

■ REST

REST, Representational State Transfer, ist ein Architekturstil für das World Wide Web, den die Rails-Entwickler Ansätzen wie SOAP, WSDL usw. vorziehen. Rails bietet eine hervorragende Unterstützung für REST mit vielen Best Practices, die das Leben des Rails-Entwicklers weiter vereinfachen. Ein Verständnis der konzeptionellen Hintergründe von REST ist dabei nicht zwingend notwendig, aber sehr hilfreich.

■ Feedback

Die Entwicklung von Rails-Anwendungen ist von unmittelbarem Feedback auf verschiedenen Ebenen geprägt.

Weil das Framework in Ruby geschrieben wurde, erhalten Sie Feedback schon während der Entwicklung. Die Prinzipien *Konvention über Konfiguration* und *DRY* ermöglichen ein effizientes Entwickeln und fördern das schnelle Feedback. Die konsequente Unterstützung für Unit Tests in allen MVC-Schichten bietet Ihnen ebenfalls ein unmittelbares Feedback bei der Entwicklung.

Dank Scaffolding ist eine ständig lauffähige Version der Anwendung vorhanden. Diskussionen mit dem Kunden über fachliche Anforderungen, Ablauf, Masken und anderes erfolgt an einem laufenden System und nicht in der Theorie oder auf dem Papier. Die Anforderungen und Wünsche können so präziser erörtert werden. Sie bekommen schneller Feedback. Die schrittweise Erweiterung des lauffähigen Systems unterstützt sehr kurze Iterationen und damit wiederum schnelles Feedback.

■ Konzentration auf Geschäftslogik

Rails nimmt Ihnen viele technische Details ab. Statt das Framework mit Konfiguration und Code zu versorgen, konzentrieren Sie sich auf die Umsetzung der Geschäftslogik. Sie „leiten“ das Framework in die von Ihnen gewünschte Richtung.

■ Agilität

Rails ermöglicht Ihnen die einfache Testbarkeit, gar keine bis geringe Konfigurationsanpassungen, keinen redundanten Code, schnelle Entwicklungszyklen und unmittelbares Feedback. Wenn Sie diese Möglichkeiten nutzen, ist Ihre Anwendung jederzeit auf Änderungen vorbereitet. Rails unterstützt somit ideal die agile Softwareentwicklung.

■ Kosten

Alle aufgeführten Punkte führen zu einer effizienteren Entwicklung Ihrer Anwendungen und damit zur Reduzierung von Kosten. Sie erreichen Ihren Return of Investment deutlich schneller.

2.2 Bestandteile von Rails

In diesem Abschnitt liefern wir Ihnen einen ersten Überblick über die Bestandteile von Rails.

2.2.1 Komponenten und Zusammenspiel

Abbildung 2.1 zeigt die Komponenten von Rails und ihr Zusammenspiel, das wir kurz beschreiben wollen.

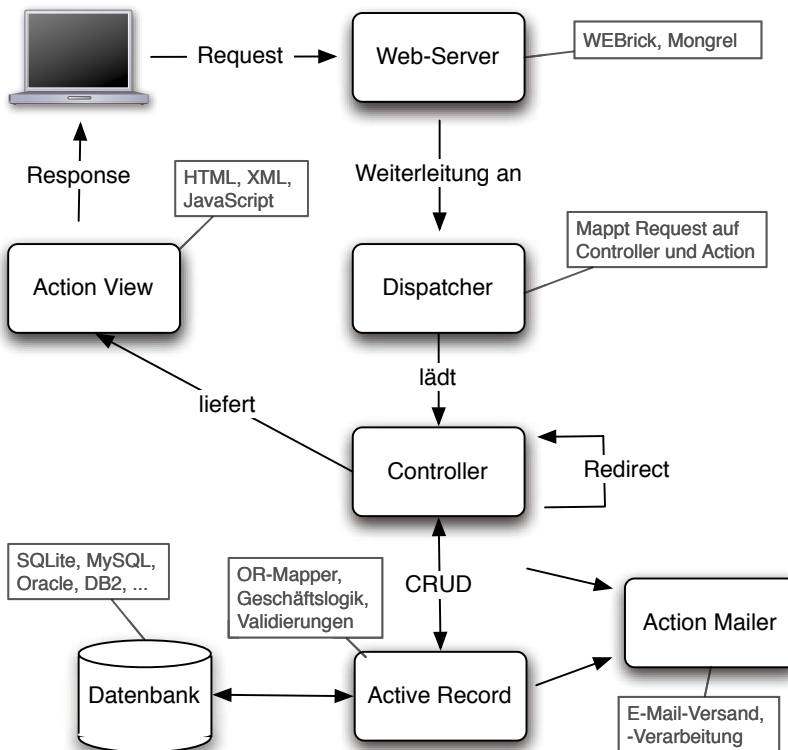


Abbildung 2.1: Bestandteile und Zusammenspiel

Der vom Client gesendete Request wird zunächst vom HTTP-Server entgegengenommen, z.B. WEBrick oder Mongrel. Der Server leitet den Aufruf an den Dispatcher weiter, bei dem es sich um ein Ruby-Programm handelt, das in jedem Rails-Projekt enthalten ist. Der Dispatcher delegiert die Anfrage an den dafür zuständigen Controller. In Rails basiert jeder Controller auf dem Framework *Action Controller* und führt typischerweise Aktionen wie Erzeugen oder Lesen auf einem Domain-Objekt aus. Dieses verwendet für seine Arbeit das *Active-Record*-Framework aus Rails, welches u.a. die Verbindung zu den Datenbanktabellen bereit-

stellt. Das Domain-Objekt validiert ggf. auch die übergebenen Parameter und liefert entsprechende Fehlermeldungen.

Der Controller führt im Anschluss eine Weiterleitung auf einen anderen Controller aus oder beginnt mit der Auslieferung der Antwort. Eine Antwort des Controllers besteht in der Regel aus einem HTML View, der mittels *Action View* aus HTML Templates erzeugt wurde. Optional versendet der Controller oder das Modell mit Hilfe des *Action Mailer*-Frameworks E-Mails.

In den folgenden Abschnitten beschreiben wir kurz die einzelnen Komponenten von Rails.

2.2.2 Action Pack

Der Name *Action Pack* steht für die Kombination der Rails-Frameworks *Action Controller* und *Action View*. *Action Controller* übernimmt dabei den Teil des Controllers, der den Request entgegennimmt und einen View als Response liefert. Es ist somit für die Logik der Verarbeitung zuständig und steht für das C im MVC-Muster.

Action View steht für das V in MVC und ist für die Repräsentation der Daten zuständig. Views werden dabei über Template-Dateien definiert, die neben HTML auch eingebetteten Ruby-Code enthalten.

Die Verarbeitung eines Requests durch Action Controller und die Erzeugung eines Views durch Action View wird in Rails als *Action* bezeichnet (daher auch der Name der Module). Eine Action wird als öffentliche Methode auf einer Controllerklasse implementiert. Ein *AddressController* stellt z.B. Methoden wie *list* oder *edit* als Actions zur Verfügung. Typischerweise werden durch Actions Domain-Objekte (z.B. eine Adresse) erzeugt, gelesen, aktualisiert oder gelöscht. Als Ergebnis liefert die Action einen View oder führt eine Weiterleitung auf eine andere Action aus.

Action Pack bietet Ihnen reichlich Unterstützung für die Entwicklung von Controllern und Views. In den Kapiteln 5 und 6 stellen wir Ihnen diese ausführlich vor.

2.2.3 Active Record

Das Framework *Active Record* repräsentiert das M im MVC-Muster und stellt die Verbindung zwischen Datenbank und Domain-Objekten her. Domain-Objekte werden in Rails als Modelle bezeichnet und durch jeweils eine Ruby-Klasse repräsentiert.

Wie wir in Kapitel 4 beschreiben werden, folgen Modelle dem Active Record-Muster aus [7], d.h. ein Modell ist mit genau einer Datenbanktabelle assoziiert. Dabei werden für die Attribute aus der Datenbanktabelle weder Getter- noch Setter-Methoden explizit im Modell definiert; diese stellt Rails für jedes Attribut aus der assoziierten Tabelle automatisch zur Verfügung.

Die Assoziationen von Modellen untereinander werden in Active Record einfach durch Klassenmethoden, wie z.B. *has_many* oder *belongs_to*, ausgedrückt. Neben einem entsprechendem Attribut in der Datenbanktabelle reicht dies Rails, um die

Verbindung von Modellen zu kennen. Für den Zugriff der assoziierten Modelle untereinander stellt Active Record automatisch Methoden zur Verfügung.

2.2.4 Action Mailer

Das Framework ermöglicht das Versenden und Empfangen von E-Mails. Der Mail-Inhalt wird dabei analog den Views über eine Template-Datei definiert. Diese kann sowohl einfachen Text wie auch HTML enthalten. Als Protokolle stehen SMTP und Sendmail zur Verfügung. Der ordnungsgemäße Versand der Mail und auch der korrekte Inhalt können über Unit Tests geprüft werden. Die Verwendung von Action Mailer wird in Kapitel 9 ausführlich beschrieben.

2.2.5 Ajax

Rails ist eines der ersten Web-Frameworks (wenn nicht das erste) mit umfangreicher Ajax-Unterstützung.⁴ Ajax ermöglicht die Entwicklung „reicher“ und „hoch interaktiver“ Web-Anwendungen. Die Anwendung verhält sich sehr viel flüssiger, als Sie es von traditionellen Anwendungen gewöhnt sind, und kommt einer „Rich-Client“-Applikation schon sehr nahe. Wir werden das Thema Ajax ausführlich in Kapitel 10 vertiefen.

2.2.6 Unit Tests

Rails unterstützt das Testen Ihrer Komponente, egal ob Modell, View oder Controller. Rails stellt Ihnen ein Grundgerüst von Unit Test-Klassen und so genannte Fixtures für die Erzeugung von Testdaten zur Verfügung. Rails bietet bezüglich des Testens der Web-Anwendung eine ideale Unterstützung. In Kapitel 14 gehen wir näher darauf ein.

2.3 Installation

Rails ist plattformunabhängig und läuft daher prinzipiell auf allen Betriebssystemen, die Ruby unterstützen. Download-Links sowie aktuelle Informationen zur Installation von Ruby, Rails und weiteren Paketen finden Sie unter <http://rubyonrails.org/down>.

2.3.1 Windows

Für einen schnellen Einstieg unter Windows empfehlen wir das Installationspaket *Instant Rails*, das Sie unter <http://instantrails.rubyforge.org/wiki/wiki.pl> herunterladen können. Es enthält neben Ruby, Rails, Apache, MySQL auch ein lauffähiges Beispiel.

⁴ Abk. für Asynchronous JavaScript and XML. Siehe <http://de.wikipedia.org/wiki/AJAX>

2.3.2 Mac

Benutzer von Mac OS X 10.5 (Leopard) bekommen Rails bereits mitgeliefert. Für Benutzer von Mac OS X 10.4 oder älteren Versionen empfehlen wir die Installation über die Seite <http://hivelogic.com/articles/ruby-rails-mongrel-mysql-osx>.

2.3.3 Linux

Benutzer von Linux wissen sicher besser als wir, wie eine optimale Installation erfolgt. Die Vielfalt der Linux-Variationen können hier nicht berücksichtigt werden. Wir empfehlen Ihnen die Seite <http://hivelogic.com/articles/ruby-rails-mongrel-mysql-osx>. Hierbei handelt es sich zwar um eine Beschreibung für Mac-Benutzer, aber die Installationsschritte sind eine gute Orientierung.

2.3.4 Aktualisierung

Ein Aktualisierung der Rails-Version bzw. aller Ruby-Bibliotheken erfolgt über den Paketmanager *RubyGems* (<http://docs.rubygems.org/>). Dieser enthält das Programm *gem*, das nach einer erfolgreichen Installation von Ruby und Rails auf dem Rechner vorhanden ist. Die Aktualisierung einer bestehenden Rails-Version erfolgt auf der Kommandozeile per:

```
$ gem update rails
```

Eine Aktualisierung der RubyGems-Installation selbst nehmen Sie wie folgt vor:

```
$ gem update --system
```

Machen Sie sich mit dem Programm etwas vertraut.

2.3.5 Datenbank

Für die Beispiele in diesem Buch verwenden wir MySQL als Datenbank. Informationen und ein Installationsprogramm für Windows finden Sie unter <http://www.mysql.de>, weitere Informationen über die Installation und Konfiguration anderer Datenbanken auf der Rails-Homepage <http://wiki.rubyonrails.com/rails/pages/DatabaseDrivers>.

2.3.6 Glückwunsch! Willkommen an Bord!

Sie können die Installation von Rails überprüfen, indem Sie ein erstes Testprojekt anlegen. Die Erzeugung eines Rails-Projekts erfolgt z.B. über die Kommandozeile, indem Sie den Befehl *rails*, gefolgt vom Namen des zu erstellenden Projekts, eingeben:

```
$ rails start  
  create  
  create  app/apis  
  create  app/controllers  
  ...
```

Wechseln Sie anschließend in das Verzeichnis *start* und starten Sie WEBrick⁵:

```
$ cd start
$ 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
... INFO WEBrick 1.3.1
... INFO ruby 1.8.6 (2007-09-24) [universal-darwin9.0]
... INFO WEBrick::HTTPServer#start: pid=7629 port=3000
```

Unter Windows und *InstantRails* erfolgt die Erzeugung eines Projekts und der Start des Webserver über die entsprechenden Menüpunkte.

Im Browser geben Sie die URL *http://localhost:3000/* ein. Wenn alles richtig installiert wurde, sehen Sie folgende *Rails Welcome Page* :

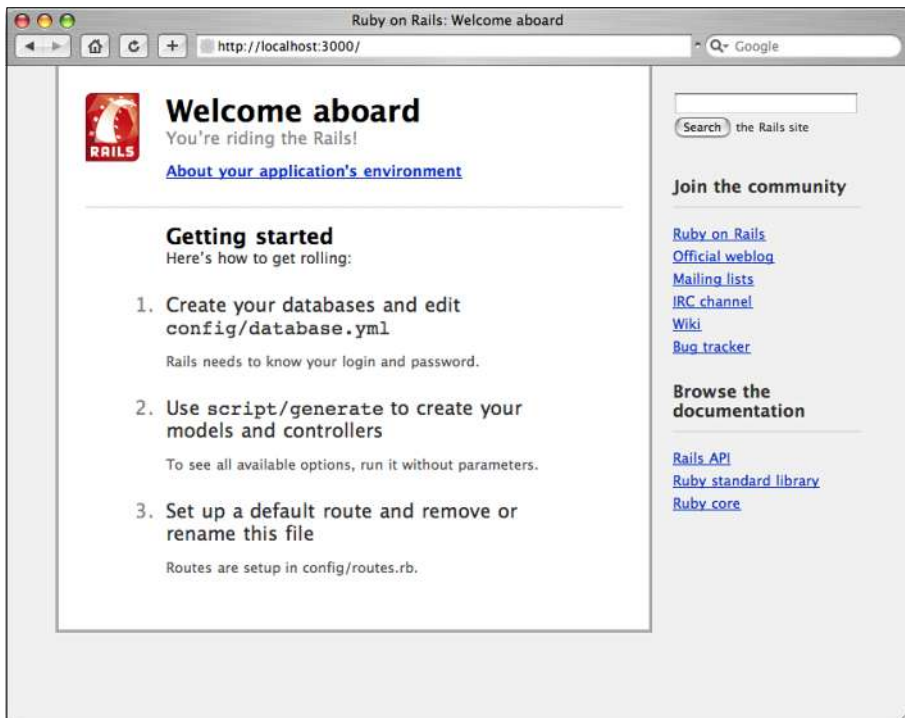


Abbildung 2.2: Rails-Willkommenseite

Herzlichen Glückwunsch! Sie haben Ruby auf die Schiene gebracht, die Reise kann beginnen!

⁵ Unter Windows `script\server`

Kapitel 3

Hands-on Rails

In diesem Kapitel entwickeln wir eine erste Web-Applikation mit Ruby on Rails. Unser Ziel ist es, Ihnen alle wesentlichen Komponenten des Frameworks und den Rails-Entwicklungsprozess im Schnelldurchlauf zu präsentieren. Dabei gehen wir nicht ins Detail, verweisen aber auf die nachfolgenden tiefer gehenden Kapitel und Abschnitte. Bei Bedarf können Sie dort nachschauen.

Rails basiert auf der Programmiersprache Ruby (siehe Kasten *Ruby*). Sollten Sie noch keine Erfahrungen mit Ruby haben, stellt dies für das Verständnis kein Problem dar. Wir liefern Ihnen an den entsprechenden Stellen die jeweils notwendigen Erklärungen in Form graphisch hervorgehobener Kästen.

Ruby

Ruby ist eine rein objektorientierte, interpretierte und dynamisch typisierte Sprache. Sie wurde bereits 1995 von Yukihiro Matsumoto entwickelt und ist neben Smalltalk und Python vor allem durch Perl beeinflusst.

Alles in Ruby ist ein Objekt, es gibt keine primitiven Typen (wie z.B. in Java). Ruby bietet neben der Objektorientierung unter anderem Garbage Collection, Ausnahmen (Exceptions), Reguläre Ausdrücke, Introspektion, Code-Blöcke als Parameter für Iteratoren und Methoden, die Erweiterung von Klassen zur Laufzeit, Threads und vieles mehr. Weitere Informationen zu Ruby finden Sie auf der Seite www.rapidwebdevelopment.de, von der Sie auch das zum Buch gehörende Ruby-Grundlagenkapitel im PDF-Format herunterladen können.

Als fachlichen Rahmen der Anwendung haben wir das Thema „Web-basiertes Projektmanagement“ gewählt, wofür es zwei Gründe gibt: Zum einen haben wir in unserem ersten Rails-Projekt eine Projektmanagement-Software entwickelt. Und zum anderen denken wir, dass viele Leser die in diesem Kapitel entwickelte Software selbst benutzen können.

Die Software ist keinesfalls vollständig, enthält jedoch alle wesentlichen Komponenten einer Web-Anwendung (Datenbank, Login, Validierung etc.). Wir denken, dass das System eine gute Basis für Weiterentwicklung und Experimente darstellt. Den

kompletten Quellcode können Sie unter www.rapidwebdevelopment.de herunterladen.

Vorweg noch eine Bemerkung zum Thema Internationalisierung: Das Beispiel in diesem Kapitel wird zur Gänze englischsprachig entwickelt. Rails beinhaltet derzeit noch keinen Standard-Mechanismus für die Internationalisierung von Web-Anwendungen. Dem Thema widmen wir uns ausführlicher in Kapitel 8.

3.1 Entwicklungsphilosophie

Bei der Entwicklung unserer Projektmanagement-Software *OnTrack* wollen wir bestimmte Grundsätze beachten, die uns auch in unseren „richtigen“ Projekten wichtig sind. Da dieses Kapitel kein Ausflug zu den Ideen der agilen Softwareentwicklung werden soll, beschränken wir uns bei der Darstellung unserer Entwicklungsphilosophie auf einen Punkt, der uns besonders am Herzen liegt: *Feedback*.

Bei der Entwicklung eines Systems wollen wir möglichst schnell Feedback bekommen. Feedback können wir dabei auf verschiedenen Ebenen einfordern, z.B. durch Unit Tests, Pair-Programming oder sehr kurze Iterationen und damit schnelle Lieferungen an unsere Kunden. Bei der Entwicklung von *OnTrack* konzentrieren wir uns auf den zuletzt genannten Punkt:¹ kurze Iterationen und schnelle Lieferung.

Geleitet von diesem Grundsatz müssen wir sehr schnell einen funktionstüchtigen Anwendungskern entwickeln, den wir unseren Kunden für die Tests zur Verfügung stellen, um von ihnen Feedback zu bekommen. Themen wie „Layouting“ oder auch eher technische Themen wie „Login“ oder „Internationalisierung“ spielen deshalb zunächst eine untergeordnete Rolle, da sie wenig mit der Funktionsweise des eigentlichen Systems zu tun haben und deshalb wenig Feedback versprechen.

3.2 Domain-Modell

Wir starten unser erstes Rails-Projekt mit der Entwicklung eines Domain-Modells. Ziel dabei ist, das Vokabular der Anwendung zu definieren und einen Überblick über die zentralen Entitäten des Systems zu bekommen.

Unser Domain-Modell besteht im Kern aus den Klassen *Project*, *Iteration*, *Task* und *Person*. *Project* modelliert die Projekte des Systems. Eine *Iteration* ist eine zeitlich terminierte Entwicklungsphase, an deren Ende ein potenziell benutzbares System steht. Iterationen stehen in einer N:1-Beziehung zu Projekten, d.h. ein Projekt kann beliebig viele Iterationen haben.

Die eigentlichen Aufgaben eines Projekts werden durch die Klasse *Task* modelliert. Tasks werden auf Iterationen verteilt, d.h. auch hier haben wir eine N:1-Beziehung zwischen Tasks und Iterationen. Bleibt noch die Klasse *Person*, die die Benutzer un-

¹ Als Anhänger der testgetriebenen Entwicklung entwickeln wir für unsere Anwendungen eigentlich immer zuerst Unit Tests. Wir haben uns aber entschieden, das Thema „Testen“ in den Kapiteln 13 und 14 gesondert zu behandeln, um Kapitel 3 nicht ausufern zu lassen.

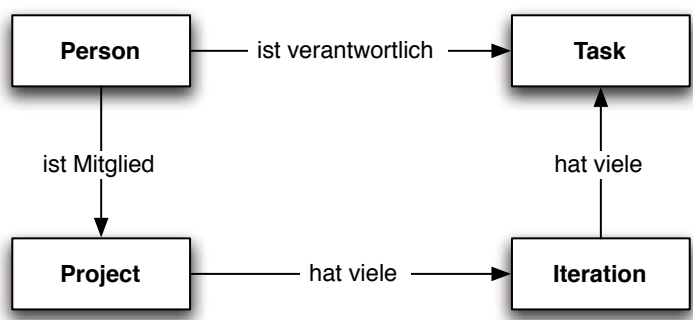


Abbildung 3.1: OnTrack-Domain-Modell

seres Systems modelliert. Die Klasse dient zum einen als Basis für die Benutzerverwaltung, zum anderen aber auch zur Verwaltung der Mitglieder eines Projekts. Das beschriebene Domain-Modell ist keinesfalls vollständig, sondern sollte eher als eine Art Startpunkt der Entwicklung gesehen werden.

3.3 OnTrack Product Backlog

Wir verwalten die Anforderungen unseres Systems in einem Product Backlog². Dies ist eine nach Prioritäten sortierte Liste von einzelnen Anforderungen (Backlog Items), die jede für sich genommen einen Mehrwert für die Benutzer des Systems darstellen. Die Priorität der Anforderung gibt die Reihenfolge ihrer Bearbeitung vor, sodass immer klar ist, was es als Nächstes zu tun gibt.

Tabelle 3.1: OnTrack 1.0 Product Backlog

Backlog-Item	Priorität
Aufsetzen der Infrastruktur	1
Projekte erfassen, bearbeiten und löschen	1
Iterationen hinzufügen	1
Iterationen anzeigen, bearbeiten und löschen	1
Tasks hinzufügen	1
Tasks anzeigen, bearbeiten und löschen	1
Struktur in die Seiten bringen	2
Validierung	2
User Login bereitstellen	2
Verantwortlichkeiten für Tasks vergeben	2

Um möglichst früh Feedback von unseren Kunden zu bekommen, müssen wir sehr schnell eine benutzbare Anwendung entwickeln, die alle notwendigen Kernfunktio-

² Product Backlogs sind ein von Ken Schwaber eingeführtes Konzept zur Verwaltung von Anforderungen im Rahmen des Scrum-Prozesses. Interessierte Leser finden eine gute Einführung in Scrum in [6].

nen zur Verfügung stellt. Deshalb haben alle Items, die für die initiale Benutzbarkeit des Systems wichtig sind, die Priorität 1 bekommen.

3.4 Aufsetzen der Infrastruktur

Jedes Rails-Projekt startet mit dem Aufsetzen der Infrastruktur. Das ist eigentlich so einfach, dass der Eintrag ins Backlog fast länger dauert als die eigentliche Aufgabe. Wir generieren unseren Anwendungsrahmen durch Ausführung des Kommandos *rails* und wechseln in das Projektverzeichnis *ontrack*:

```
$ rails ontrack
  create
  create  app/controllers
  create  app/helpers
  ...
$ cd ontrack/
```

Als Nächstes konfigurieren wir die Datenbankverbindung, indem wir die Datei *config/database.yml*³ (siehe Kasten *YAML*) editieren und die entsprechenden Verbindungsdaten eintragen.

YAML

YAML (YAML Ain't Markup Language) ist ein einfaches Format für die Serialisierung und den Austausch von Daten zwischen Programmen. Es ist von Menschen lesbar und kann leicht durch Skripte verarbeitet werden. Ruby enthält ab Version 1.8.0 eine YAML-Implementierung in der Standard-Bibliothek. Rails nutzt das Format für die Datenbankkonfiguration und Unit Test Fixtures (siehe Kapitel 14). Weitere Infos finden Sie unter <http://www.yaml.org>

Die Datei enthält Default-Einstellungen für drei Datenbanken: *development* für die Entwicklungsphase des Systems, *test* für automatisierte Unit Tests (siehe auch Abschnitt 14.2) und *production* für die Produktionsversion der Anwendung (siehe dazu Abschnitt 11.1).

Uns interessiert für den Moment nur die Entwicklungsdatenbank, die Rails per Konvention mit dem Präfix des Projekts und dem Suffix *development* benennt:

Listing 3.1: config/database.yml

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  timeout: 5000
test:
  ...
```

³ Die im Folgenden verwendeten Verzeichnisnamen beziehen sich immer relativ auf das Root-Verzeichnis der Anwendung *ontrack*.

```
production:
  ...
```

Rails erstellt per Default einen Eintrag für die SQLite-Datenbank.⁴ Wenn Sie eine andere Datenbank verwenden möchten, müssen Sie den Konfigurationseintrag entsprechend ändern. Alternativ können Sie die Datenbank auch bei der Erzeugung des Projekts angeben, z.B.:

```
$ rails -d mysql ontrack
```

Eine Liste von unterstützten Datenbanken finden Sie auf der Rails-Homepage⁵. Wir verwenden für die OnTrack-Anwendung eine MySQL⁶-Datenbank, deren Konfiguration wie folgt aussieht. Alle anderen Einstellungen sowie die Einstellungen der beiden Datenbanken *test* und *production* lassen wir zunächst unverändert:

Listing 3.2: config/database.yml

```
development:
  adapter: mysql
  database: ontrack_development
  host: localhost
  username: root
  password:

test:
  ...
production:
  ...
```

Nachdem wir die Datenbank konfiguriert haben, müssen wir sie natürlich auch erstellen. Rails bietet dazu den *rake*-Task *db:create* (siehe auch Kasten *rake*), der wie folgt ausgeführt wird:

```
$ rake db:create
```

Alternativ kann die Datenbank auch per *mysql*-Client angelegt werden:

```
$ mysql -u root
mysql> create database ontrack_development;
```

Zum Abschluss fehlt noch der Start des HTTP-Servers. Wir verwenden in unserer Entwicklungsumgebung den in Ruby programmierten HTTP-Server *WEBrick*. Er benötigt keine Konfiguration und ist durch seine leichte Handhabung gerade für die Entwicklung geeignet. Alternativ können Sie auch Mongrel⁷ (vgl. 11.2.3) verwenden. Zum Starten des Servers geben wir den Befehl *ruby script/server webrick*⁸ ein:

⁴ Siehe <http://www.sqlite.org/>

⁵ <http://wiki.rubyonrails.com/rails/pages/DatabaseDrivers>

⁶ Siehe <http://www.mysql.de>

⁷ <http://mongrel.rubyforge.org/>

⁸ Die Skripte einer Rails-Anwendung liegen im Verzeichnis *APP_ROOT/script*.

```
$ ruby script/server webrick
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
... WEBrick 1.3.1
... ruby 1.8.4 (2005-12-24) [powerpc-darwin8.6.1]
... WEBrick::HTTPServer#start: pid=3419 port=3000
```

Ruby: rake

Das Programm *rake* dient zur Ausführung definierter Tasks analog dem Programm *make* in C oder *ant* in Java. Rails definiert bereits eine Reihe von Tasks. So startet z.B. der Aufruf von *rake* ohne Parameter alle Tests zum Projekt, oder *rake stats* liefert eine kleine Projektstatistik. Das Programm wird uns im Laufe des Buches noch häufiger begegnen. Weitere Infos finden sich unter <http://rake.rubyforge.org>.

3.5 Projekte erfassen, bearbeiten und löschen

Eine Projektmanagement-Software ohne Funktion zur Erfassung von Projekten ist wenig sinnvoll. Deshalb steht die Erfassung und Bearbeitung von Projekten auch ganz oben in unserem Backlog.

3.5.1 Modell erzeugen

Für die Verwaltung von Projekten benötigen wir die Rails-Modellklasse *Project*. Modellklassen sind einfache Domain-Klassen, d.h. Klassen, die eine Entität der Anwendungsdomäne modellieren. Neben der Modellklasse benötigen wir einen Controller, der den Kontrollfluss unserer Anwendung steuert. Den Themen Modelle und Controller widmen wir uns ausführlich in den Kapiteln 4 und 5.

Modelle und Controller werden in Rails-Anwendungen initial generiert. Hierfür liefert Rails das Generatorprogramm *generate*, das wir wie folgt aufrufen:

```
$ ruby script/generate scaffold project name:string \
  description:text start_date:date
```

Der erste Parameter *scaffold* gibt den Namen des Generators und der zweite Parameter *project* den Namen der Modellklasse an. Der Modellklasse folgt eine optionale Liste von Modellattributen, d.h. Datenfelder, die das erzeugte Modell besitzen soll. Die Angabe der Modellattribute hat in der Form *Attributname:Attributtyp* zu erfolgen. Die Ausgabe des Generators sieht in etwa wie folgt aus:

```
$ ruby script/generate scaffold project name:string \
  description:text start_date:date
  exists  app/models/
  exists  app/controllers/
  exists  app/helpers/
  create  app/views/projects
```

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

Wir verwenden in unserem Beispiel den Scaffold⁹-Generator, der neben dem eigentlichen Modell einen Controller sowie die zugehörigen HTML-Seiten erzeugt. Doch konzentrieren wir uns zunächst auf die Modellklasse.

Neben dem Modell selbst (*app/models/project.rb*) werden ein Unit Test (*test/unit/project_test.rb*), eine Fixture-Datei mit Testdaten (*test/fixtures/projects.yml*) und ein Migrationsskript (*db/migrate/001_create_projects.rb*) erzeugt. Das Thema „Testen“ behandeln wir ausführlich in Kapitel 13 und 14. Im Moment interessieren uns nur das Modell und das Migrationsskript.

Der Generator erzeugt eine zunächst leere Modellklasse (siehe Kasten *Klasse*). Jedes Modell erbt in Rails von der Klasse *ActiveRecord::Base*, auf die wir im Kapitel 4 genauer eingehen:

Listing 3.3: *app/models/project.rb*

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

Wie wir noch sehen werden, ist das Modell dank der Vererbung von *ActiveRecord::Base* aber voll einsatzfähig und muss zunächst nicht erweitert werden. Schauen wir uns daher das Migrationsskript an.

⁹ Zu Deutsch so viel wie Gerüstbau

Ruby: Klasse

Eine Klasse wird in Ruby durch das Schlüsselwort *class* eingeleitet. Die Vererbung wird durch *<*, gefolgt von der Superklasse, definiert. Im Listing 3.3 erbt die Klasse *Project* somit von der Klasse *Base*, die im Namensraum *ActiveRecord* definiert ist. Namensräume werden in Ruby durch Module definiert.

3.5.2 Datenbankmigration

Datenbankänderungen werden in Rails nicht in SQL programmiert, sondern über so genannte Migrationsskripte in Ruby. Diese haben den Vorteil, dass die Schemata inklusive Daten in Ruby und damit Datenbank-unabhängig vorliegen. Ein Wechsel der Datenbank wird somit erleichtert, z.B. von SQLite während der Entwicklung zu MySQL in Produktion oder von Ralf mit MySQL zu Thomas mit Oracle.

Ebenso wird die Teamarbeit an unterschiedlichen Versionsständen des Projekts unterstützt. Zu einem Release oder einer Version gehören auch die Migrationsskripte, die die Datenbank mit Entwicklungs- und Testdaten auf den zur Software passenden Stand bringen. Und zwar mit einem Befehl. Vorbei sind die Zeiten umständlichen Gefummels an Schemata und Daten per SQL.

Da es sich um Ruby-Programme handelt, können damit jegliche Aktionen durchgeführt werden; neben den Änderungen an Schemata und Daten z.B. auch das Lesen initialer Daten aus einer CSV-Datei oder entsprechender Code zur Migration von (Teil-)Daten aus einer Tabelle in eine andere.

Migrationsskript definieren

Wie beschrieben, erzeugt der Generator automatisch ein entsprechendes Migrationsskript zum Modell. Das liegt daran, dass jede Modellklasse eine Datenbanktabelle benötigt, in der die Instanzen dieser Modellklasse gespeichert werden (vgl. Kapitel 4).

Eine Rails-Konvention besagt, dass die Tabelle einer Modellklasse den pluralisierten Namen des Modells haben muss. Die Tabelle unserer *Project*-Klasse muss also *projects* heißen und wird vom Generator daher im Migrationsskript bereits so benannt:

Listing 3.4: db/migrate/001_create_projects.rb

```
class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      t.string :name
      t.text :description
      t.date :start_date

      t.timestamps
    end
  end
end
```

```

def self.down
  drop_table :projects
end
end

```

Bei einem Migrationsskript handelt es sich um eine Klasse, die von *ActiveRecord::Migration* erbt und die Methoden *up* und *down* implementiert. In *up* definieren Sie alle Änderungen an der Datenbank, und in *down* machen Sie diese wieder rückgängig. So können Sie mit jedem Skript eine Version vor und zurück migrieren.

Ruby: Symbole

In Ruby werden anstelle von Strings häufig Symbole verwendet. Ein Symbol wird durch einen führenden Doppelpunkt (:) gekennzeichnet. Symbole sind gegenüber Strings atomar, d.h. es gibt für ein und dasselbe Symbol nur genau eine Instanz. Egal, wo im Programm das Symbol auch referenziert wird, es handelt sich immer um dasselbe Symbol (dieselbe Instanz). Symbole werden daher dort verwendet, wo keine neue Instanz eines Strings benötigt wird. Typische Stellen sind Schlüssel in Hashes oder die Verwendung für Namen.

Eine Attributdefinition enthält einen Typ (z.B. *t.string*), gefolgt von einem oder mehreren Attributnamen. Alle weiteren Parameter sind optional und werden über eine Hash definiert (siehe Kasten *Hash*). Die Schreibweise von Namen mit einem führenden Doppelpunkt (z.B. *:project*) definiert ein Symbol (siehe Kasten *Symbol*). Eine Übersicht aller unterstützten Methoden, Datentypen und Parameter findet sich im Anhang unter dem Abschnitt 14.11.

Ruby: Hash

Eine Hash wird in Ruby typischerweise durch geschweifte Klammern {} und einen Eintrag durch *Schlüssel => Wert* erzeugt. Beispielsweise wird in der ersten Definitionszeile in Listing 3.4 eine Hash mit dem Schlüssel *:null* erzeugt und als Parameter an die Methode *t.string* übergeben. Der Einfachheit halber wurden die geschweiften Klammern weggelassen. Das wird Ihnen bei Rails häufig begegnen. Der Zugriff auf die Elemente einer Hash erfolgt über den in eckigen Klammern eingeschlossenen Schlüssel, z.B. *options[:name]*.

Hinweis: Die Schreibweise *t.string* zur Definition eines neuen Datenbankfelds ist erst seit Rails 2.0 verfügbar. Frühere Versionen von Rails verwenden stattdessen die Methode *column*, die auf dem Table-Objekt *t* aufgerufen wird. Letztere Variante erwartet die Angabe des zu erzeugenden Attributtyps als Methodenparameter:

```

class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      # in Rails < 2.0
      t.column :name, :string
      t.column :description, :text
    end
  end
end

```



```

      # in Rails >= 2.0
      t.string :name
      t.text   :name
    end
  end

  ...
end

```

Der Primärschlüssel (vgl. 4.1.2) mit dem Namen *id* wird von Rails automatisch in die Tabelle eingetragen. Der Eintrag *t.timestamps* sorgt für die Erzeugung der Tabellenattribute *created_at* und *updated_at*. Diese enthalten das Datum und die Uhrzeit der Erzeugung bzw. der Aktualisierung des Eintrags. Benötigen Sie diese Werte nicht, löschen Sie die Zeile aus dem Migrationsskript.

Constraints werden in Rails typischerweise nicht auf Ebene der Datenbank definiert, sondern auf Modellebene, da sie die Datenbank-Unabhängigkeit einschränken. Sind dennoch Constraints nötig, so sind diese über die Methode *execute* per SQL zu definieren (s.u.).

Migration ausführen

Zur letzten Version migrieren wir immer durch den folgenden Aufruf mit Hilfe von *rake*:

```

$ rake db:migrate
(in .../ontrack)
== 1 CreateProjects: migrating =====
-- create_table(:projects)
   -> 0.0371s
== 1 CreateProjects: migrated (0.0373s) =====

```

Rails ermittelt hierbei aus der Tabelle *schema_info*¹⁰ die aktuelle Migrationsversion der Datenbank. Jedes Skript erhält bei der Generierung eine fortlaufende Nummer (001, 002, ...) als Präfix im Dateinamen. Auf diese Weise kann Rails entscheiden, welche Skripte aufgerufen werden müssen, bis die Version der Datenbank mit dem des letzten Skripts übereinstimmt.

Enthält die Tabelle *schema_info* z.B. den Wert 5, und das letzte Skript unter *db/migrate* beginnt mit *008_*, so wird Rails die Skripte 006, 007 und 008 und darin jeweils die Methode *up* der Reihe nach ausführen und am Ende den Wert in *schema_info* auf 8 aktualisieren.

Um zu einer konkreten Version zu migrieren, z.B. zurück zu einer älteren, geben wir die Migrationsversion über die Variable *VERSION* an:

```
$ rake db:migrate VERSION=X
```

Wollen wir z.B. von der Version 8 wieder auf 5 zurück, erhält *VERSION* den Wert 5, und Rails ruft jeweils die Methode *down* für die Skripte 008, 007 und 006 hintereinander auf.

¹⁰Diese Tabelle wird beim allerersten Aufruf von *rake migrate* automatisch erzeugt.

Hinweise zur Migration

Migrationsskripte können auch direkt über einen eigenen Generator erzeugt werden. Das Skript aus unserem Beispiel würden wir in diesem Fall wie folgt erzeugen. Beachten Sie, dass keine Nummer als Präfix angegeben wird, da diese Aufgabe der Generator übernimmt:

```
$ ruby script/generate migration create_projects
```

Sie können ebenso ein Modell ohne Migrationsskript erzeugen. Verwenden Sie hierzu beim Aufruf den Parameter `--skip-migration`:

```
$ ruby script/generate model project --skip-migration
...
create  app/models/project.rb
create  test/unit/project_test.rb
create  test/fixtures/projects.yml
```

Da es sich bei einem Migrationsskript um Ruby-Code handelt, können wir hier im Grunde alles programmieren, was für eine automatisierte Migration benötigt wird. Wir können mehr als eine Tabelle erzeugen und löschen, initiale Daten definieren, Daten von Tabellen migrieren usw. Es ist aber immer darauf zu achten, die Änderungen in `down` in umgekehrter Reihenfolge wieder rückgängig zu machen. Beachten Sie auch, dass ein Migrationsskript ggf. ein oder mehrere Modelle nutzt, weshalb diese vor dem Ausführen des Skripts existieren müssen. Sollte eine Migration einmal nicht möglich sein, wird dies durch die Ausnahme `ActiveRecord::IrreversibleMigration` wie folgt angezeigt:

Listing 3.5: db/migrate/001_create_projects.rb

```
class CreateProjects < ActiveRecord::Migration
  ..
  def self.down
    # Ein Zurück gibt es diesmal nicht.
    raise ActiveRecord::IrreversibleMigration
  end
end
```

Über die Methode `execute` kann auch direkt SQL durch das Skript ausgeführt werden. Im folgenden Beispiel ändern wir z.B. den Tabellentyp unserer MySQL-Datenbank von `MyIsam` auf `InnoDB`:

```
class SwitchToInnoDB < ActiveRecord::Migration
  def self.up
    execute "ALTER TABLE projects TYPE = InnoDB"
  end

  def self.down
    execute "ALTER TABLE projects TYPE = MyIsam"
  end
end
```

Bei der direkten Verwendung von SQL ist zu beachten, dass ggf. die Datenbankunabhängigkeit verloren geht. Möglicherweise können dann Bedingungen helfen, z.B.:

```
class SwitchToInnoDB < ActiveRecord::Migration
  def self.up
    if ENV["DB_TYPE"] == "mysql"
      execute "ALTER TABLE projects TYPE = InnoDB"
    end
  end
  ...
end
```

Bei der Ausführung des Skripts wird dann im konkreten Fall die Umgebungsvariable gesetzt:

```
$ DB_TYPE=mysql rake db:migrate
```

Kommt es während der Ausführung eines Migrationsskripts zu einem Fehler, ist eventuell etwas Handarbeit gefragt. Im einfachsten Fall korrigieren wir den Fehler und starten das Skript erneut. Möglicherweise müssen wir vorher die Versionsnummer in der Tabelle *schema_info* von Hand setzen oder auch bereits angelegte Tabellen löschen. Dies geht natürlich auch, indem wir alle temporär nicht relevanten Befehle im Skript auskommentieren und dieses starten. Wir empfehlen mit Nachdruck, für jedes neue Skript einmal vor und zurück zu migrieren, um das Skript auf diese Weise zu testen. Ihre Teamkollegen werden es Ihnen danken.

Migrationsskripte sind ein optimales Werkzeug zur Projektautomatisierung. Einmal damit vertraut gemacht, werden Sie sie nicht mehr missen wollen.

3.5.3 Controller

Neben der Modellklasse *Project* hat der Scaffold-Generator einen CRUD¹¹-Controller inklusive der zugehörigen HTML-Seiten erzeugt. Der Controller enthält vorgefertigten Quellcode zum Anlegen, Löschen und Bearbeiten von *Project*-Instanzen, so dass die Entwicklungsarbeiten für unser erstes Backlog-Item bereits abgeschlossen sind und die Anwendung gestartet werden kann. Öffnen Sie die Startseite <http://localhost:3000/projects> der Anwendung in einem Browser, und prüfen Sie es selbst. Schon jetzt stehen Ihnen die Seiten aus Abbildung 3.2 zur Verfügung.

¹¹ Das Akronym CRUD steht für *create*, *read*, *update* und *delete*.

Ein Hinweis zu REST

Mit Version 2.0 setzt Rails auf REST als Standard-Entwicklungsparadigma. Teil der REST-Philosophie in Rails ist unter anderem die Verwendung von REST-basierten CRUD-Controllern, d.h. für jedes Modell ist genau ein Controller zuständig. Dies gilt beispielsweise für den zuvor generierten *ProjectsController*. Wir weichen in diesem Abschnitt an einigen Stellen bewusst von dieser Philosophie ab, da wir das Kapitel einfach halten und neben elementaren Rails-Grundlagen nicht zusätzlich REST einführen wollen. Stattdessen widmen wir dem Thema REST ein eigenes Kapitel (siehe Kapitel 7: *RESTful Rails*), in dem wir die hier gelegten Grundlagen aufgreifen und REST-basiert neu entwickeln.

Bereits nach wenigen Handgriffen können Projekte angelegt, bearbeitet und gelöscht werden. Beachten Sie, dass wir dafür kaum eine Zeile Code geschrieben und die Anwendung weder kompiliert noch deployed haben. Sie haben einen ersten Eindruck der Möglichkeiten von Rails erhalten. Das macht Lust auf mehr, nicht wahr?!

Aber keine Sorge, Rails ist kein reines Generator-Framework. Das in diesem Abschnitt beschriebene Scaffolding ist nur der Einstieg, der eine erste Version der Anwendung generiert und so die Erzeugung und Bearbeitung von Modellen eines bestimmten Typs ermöglicht. In der Praxis wird man Scaffold-Code nach und nach durch eigenen Code ersetzen. Dabei bleibt das System zu jedem Zeitpunkt vollständig lauffähig und benutzbar, da immer nur ein Teil der Anwendung (z.B. eine HTML-Seite) ersetzt wird, die anderen Systemteile aber auf Grund des Scaffold-Codes weiterhin funktionstüchtig bleiben.

Fassen wir die Schritte noch einmal kurz zusammen:

1. Erzeugung des Modells, Migrationsskripts und Controllers per Scaffold-Generator: *ruby script/generate scaffold MODEL*.
2. Definition der Datenbanktabelle etc. im Migrationsskript und Datenbankaktualisierung per *rake db:migrate*.
3. Aufruf *http://localhost:3000/CONTROLLER* und sich freuen.

3.6 Iterationen hinzufügen

Im nächsten Schritt wird die Anwendung um eine Funktion zur Erfassung von Iterationen erweitert. Jede Iteration gehört zu genau einem Projekt, und ein Projekt kann mehrere Iterationen besitzen. Die N:1-Relation müssen wir nun auf Modell- und Datenbankebene abbilden. Als Erstes benötigen wir dafür eine neue Modellklasse *Iteration*, die wir wie bekannt erzeugen:

```
$ ruby script/generate model iteration name:string \
  description:text start_date:date end_date:date \
  project_id:integer
exists app/models/
```

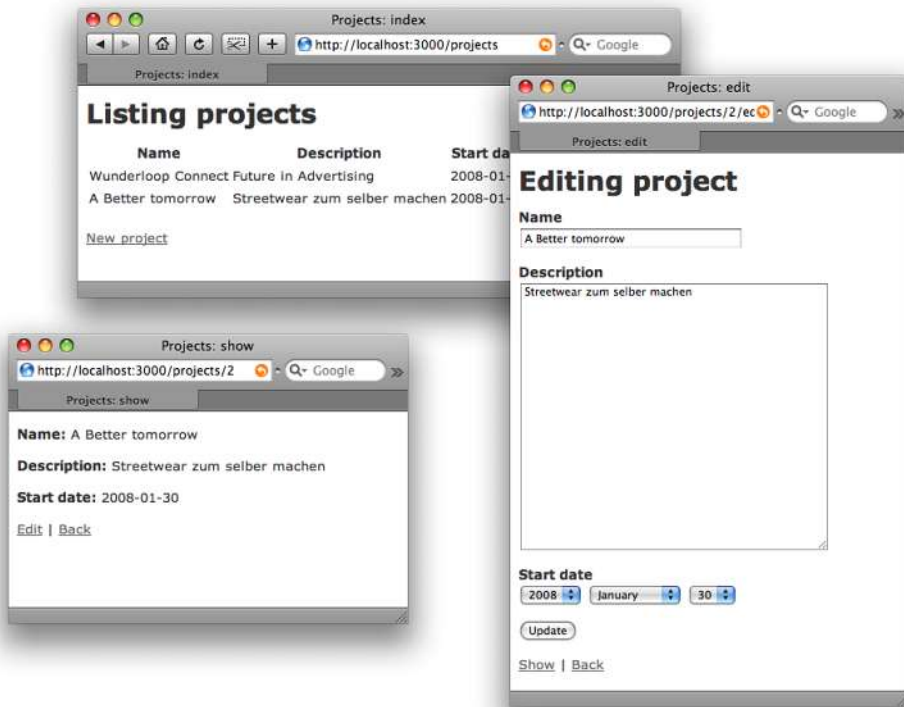


Abbildung 3.2: OnTrack in Version 0.1

```
exists test/unit/
exists test/fixtures/
create app/models/iteration.rb
create test/unit/iteration_test.rb
create test/fixtures/iterations.yml
exists db/migrate
create db/migrate/002_create_iterations.rb
```

Anschließend bringen wir die Datenbank auf den neuesten Stand. Bereits erstellte Projekte werden dabei nicht gelöscht:

```
$ rake db:migrate
(in ../ontrack)
== 2 CreateIterations: migrating =====
-- create_table(:iterations)
   -> 0.0043s
== 2 CreateIterations: migrated (0.0044s) =====
```

Das bei der Erzeugung des Modells *Iteration* definierte Feld *project.id* modelliert die N:1-Beziehung auf Datenbankebene. Es definiert den Fremdschlüssel, der zu einer Iteration die ID des zugehörigen Projekts referenziert.

Ruby: Klassenmethode

Eine Klassenmethode lässt sich direkt in der Klassendefinition aufrufen. Bei der Definition und beim Aufruf von Methoden können die Klammern weggelassen werden, sofern der Interpreter den Ausdruck auch ohne versteht. Der Methodenaufruf in Listing 3.6 ist somit eine vereinfachte Schreibweise von `belongs_to(:project)`. Durch das Weglassen der Klammern sieht der Ausdruck mehr wie eine Definition aus und weniger wie ein Methodenaufruf. Wenn Sie so wollen, *definieren* Sie die Relation und programmieren sie nicht.

Zusätzlich zum Datenmodell muss die N:1-Relation auch auf der Modellebene modelliert werden. Dazu wird die neue Klasse *Iteration* um einen Aufruf der Klassenmethode *belongs_to* erweitert (siehe Kasten *Klassenmethode*). Ihr wird der Name des assoziierten Modells übergeben:

Listing 3.6: `app/models/iteration.rb`

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

Das Hinzufügen von Iterationen zu einem Projekt soll möglichst einfach sein. Wir denken, die einfachste Möglichkeit ist die Erweiterung des List-Views für Projekte um einen neuen Link *Add Iteration* (siehe Abbildung 3.3).

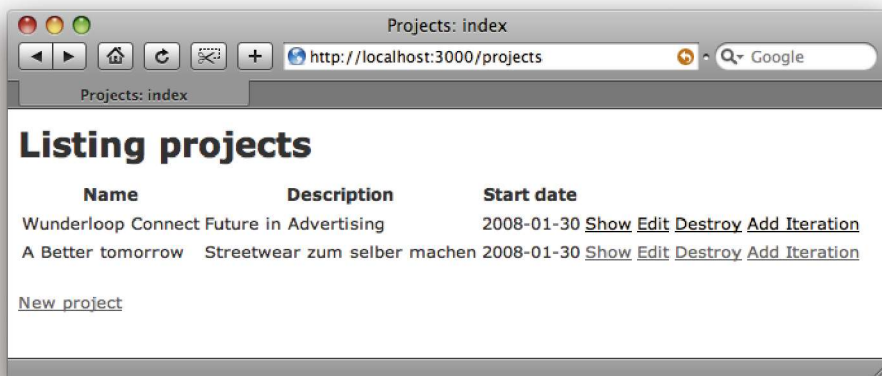


Abbildung 3.3: Ein neuer Link zum Hinzufügen von Iterationen

Views sind in Rails HTML-Seiten, die eingebetteten Ruby-Code enthalten. Wir werden darauf in Kapitel 6 eingehen. Ruby-Ausdrücke werden im HTML-Code von den Tags `<%` und `%>` eingeschlossen. Folgt dem öffnenden Tag ein `=`-Zeichen, wird das Ergebnis des enthaltenen Ausdrucks in einen String umgewandelt und in der HTML-Seite ausgegeben.

Zum Erstellen des *Add Iteration*-Links erweitern Sie den Scaffold-generierten Index-View um einen Aufruf des URL-Helpers *link_to*:

Listing 3.7: `app/views/projects/index.html.erb`

```
...
<td>
  <%= link_to "Destroy", project, :confirm => "Are you sure?",
                                :method => :delete %>

</td>
<td>
  <%= link_to "Add Iteration", :action => "add_iteration",
                                :id => project %>

</td>
...
```

URL-Helper sind Methoden, die Ihnen in jedem View zur Verfügung stehen und den HTML-Code kurz und übersichtlich halten. Rails stellt eine ganze Reihe solcher Hilfsmethoden zur Verfügung, und Sie können beliebige hinzufügen. Mehr zu URL- und Formular-Helper erfahren Sie in den Abschnitten 6.2 und 6.3.

Der URL-Helper *link_to* erzeugt einen neuen HTML-Link. Die Methode erwartet als ersten Parameter einen String mit dem Namen des Links, der im Browser erscheinen soll (z.B. *Add Iteration*). Als zweiter Parameter ist eine Hash anzugeben, deren Werte die aufzurufenden URL definieren.

Über *:action* wird die Controller-Action bestimmt, die bei der Ausführung des Links aufgerufen wird. Eine Action ist eine öffentliche Methode der Controllerklasse, die für die Bearbeitung eines bestimmten HTTP-Requests zuständig ist. Der Parameter *:id* gibt die ID des Projekts an, unter dem die neue Iteration angelegt werden soll, und wird der Action *add_iteration* übergeben.

Da wir in unserem neuen Link die Action *add_iteration* referenzieren, müssen wir die Klasse *ProjectsController* um eine entsprechende Action, d.h. um eine gleichnamige Methode erweitern (siehe Kasten Methodendefinition):

Listing 3.8: `app/controllers/projects_controller.rb`

```
class ProjectsController < ApplicationController
  def add_iteration
    project = Project.find(params[:id])
    @iteration = Iteration.new(:project => project)
    render :template => "iterations/edit"
  end
  ...
end
```

Ruby: Methodendefinition

Eine Methode wird durch das Schlüsselwort *def* eingeleitet und mit *end* beendet. Die runden Klammern können weggelassen werden. Eine Methode liefert als Rückgabewert immer das Ergebnis des zuletzt ausgewerteten Ausdrucks, sodass die explizite Angabe einer *return*-Anweisung entfallen kann.

Eine Action hat über die Methode *params* Zugriff auf die Parameter eines HTTP-Requests (vgl. Kapitel 5). In unserem Beispiel übergibt der Link *Add Iteration* der Action die Projekt-ID über den Parameter *:id*. Die Action lädt das Projekt aus der Datenbank, indem sie die statische Finder-Methode *Project.find* aufruft und die ID übergibt.

Ruby: Instanzvariablen

Eine Instanzvariable wird durch einen führenden Klammeraffen @ definiert. Instanzvariablen werden beim ersten Auftreten in einer Instanzmethode erzeugt und nicht, wie z.B. in Java, explizit in der Klasse definiert.

Anschließend wird eine neue Iteration erzeugt und der Instanzvariablen *@iteration* zugewiesen. Die neue Iteration bekommt in ihrem Konstruktor das zuvor geladene Projekt übergeben. Abschließend erzeugt die Action aus dem Template *iterations/edit* den View, der als Ergebnis zurückgeliefert wird. Dazu ist als Nächstes das Template *edit.html.erb* im Verzeichnis *app/views/iterations/* zu erzeugen:

Listing 3.9: *app/views/iterations/edit.html.erb*

```
<h1>Editing Iteration</h1>
<% form_tag :action => "update_iteration",
           :id => @iteration do %>
  <p>Name: <%= text_field "iteration", "name" %></p>
  <p>Start Date:
    <%= date_select "iteration", "start_date" %></p>
  <p>End Date:
    <%= date_select "iteration", "end_date" %></p>

  <%= submit_tag "Update" %>
  <%= hidden_field "iteration", "project_id" %>
<% end -%>
<%= link_to "Back", :action => "index" %>
```

Der Formular-Helper *form_tag* erzeugt das HTML-Element *form*. Die Parameter *:action* und *:id* geben an, welche Controller-Action beim Abschicken des Formulars aufgerufen wird und welche zusätzlichen Parameter dieser Action übergeben werden. Beachten Sie bitte, dass wir in einem View Zugriff auf die Instanzvariablen des Controllers haben. Der *ProjectsController* hat die Instanzvariable *@iteration* in der Action *add_iteration* erzeugt. Diese wird im Edit-View direkt genutzt, um darin die Daten aus dem Formular zu speichern.

Die Formular-Helper *text_field* und *date_select* erzeugen Eingabefelder für die Attribute einer Iteration. Interessant sind die Parameter dieser Methoden: Der erste Parameter *iteration* referenziert dabei das Objekt *@iteration*, welches in der Controller-Action *add_iteration* als Instanzvariable erzeugt wurde. Der zweite Parameter (z.B. *name*) gibt die Methode an, die auf dem Objekt *@iteration* aufgerufen wird, um das entsprechende Eingabefeld mit Daten vorzubelegen.

Der Formular-Helper *submit_tag* erzeugt einen Submit-Button zum Abschicken des Formulars. Der Helper *hidden_field* erzeugt ein Hidden-Field, das für die Übertra-

gung der Projekt-ID an den Server benötigt wird. Der neue View ist in Abbildung 3.4 dargestellt.



Abbildung 3.4: Ein View zum Bearbeiten von Iterationen

Der Edit-View gibt als Submit-Action *update_iteration* an, die wir im *ProjectsController* implementieren:

Listing 3.10: *app/controllers/projects_controller.rb*

```
def update_iteration
  @iteration = Iteration.new(params[:iteration])
  if @iteration.save
    flash[:notice] = "Iteration was successfully created."
    redirect_to(projects_url)
  else
    render :template => "iterations/edit"
  end
end
```

Auch diese Action verwendet die *params*-Hash zum Zugriff auf die Request-Parameter. Allerdings werden hier sämtliche Parameter auf einen Schlag geholt und der neuen Iteration an den Konstruktor übergeben. Über die *If*-Bedingung wird geprüft, ob die Iteration per *@iteration.save* erfolgreich gespeichert wurde oder nicht (siehe Kasten *If*-Anweisung).

Wenn ja, dann erfolgt eine Weiterleitung auf die Action *index*, welche eine Liste aller Projekte anzeigt. Dazu wird die Methode *redirect.to* verwendet (vgl. 5.3). Im Fehlerfall wird der Edit-View erneut angezeigt. Dieser Punkt ist für das Thema „Validierung“ von Bedeutung, mit dem wir uns in Abschnitt 3.13 genauer beschäftigen werden.

Ruby: If-Anweisung

Die *If*-Anweisung beginnt in Ruby mit dem Schlüsselwort *if* und endet mit *end*. Optional können ein oder mehrere *elsif*-Bedingungen und eine *else*-Anweisung aufgeführt werden. Die Bedingung gilt als wahr, wenn der Ausdruck einen Wert ungleich *nil* (nicht definiert) oder *false* liefert. Die explizite Prüfung auf *!= nil* kann entfallen. Neben *if* wird häufig *unless* verwendet, die elegantere Form von *if not* bzw. *if !*.

Um zu testen, ob das Hinzufügen von Iterationen funktioniert, erweitern wir den List-View für Projekte um die Ausgabe der Anzahl von Iterationen pro Projekt. Dafür benötigt die Klasse *Project* eine zusätzliche *has_many*-Deklaration:

Listing 3.11: app/models/project.rb

```
class Project < ActiveRecord::Base
  has_many :iterations, :dependent => :destroy
end
```

Die Deklaration *has_many* erzeugt für die Elternklasse (hier *Project*) einer 1:N-Relation eine Reihe von Methoden, die der Elternklasse den Zugriff auf die assoziierten Kindklassen (hier *Iteration*) ermöglichen. Eine dieser Methoden ist *iterations*, die für ein Projekt eine Liste zugehöriger Iterationen liefert. Die Option *:dependent => :destroy* sorgt für das automatische Löschen aller Kindobjekte, wenn das Elternobjekt gelöscht wird. Der folgende Code-Auszug zeigt die Verwendung der Methode *iterations* im List-View für Projekte:

Listing 3.12: app/views/projects/index.html.erb

```
<table>
  <tr>
    ...
    <th>Iterations</th>
  </tr>

  <% for project in @projects %>
    <tr>
      <td><%=h project.name %></td>
      <td><%=h project.description %></td>
      <td><%=h project.start_date %></td>
      <td><%= project.iterations.length %></td>
      ...
    <% end -%>
  </table>
  ...
```

Der vom Generator stammende Code erzeugte ursprünglich den View aus Abbildung 3.3. Für jedes Projekt aus der Liste werden Name, Beschreibung und Startdatum ausgegeben. Dabei ist die Methode *h* ein Alias für *html_escape* und konvertiert HTML-Elemente, z.B. *<* in *<*; (siehe Abschnitt 6.1).

Das obige Codebeispiel erweitert die Tabelle um eine zusätzliche Spalte *Iterations*, die mit Hilfe des Aufrufs *project.iterations.length* die Anzahl der Iterationen des jeweiligen Projekts anzeigt (siehe Abbildung 7.2).

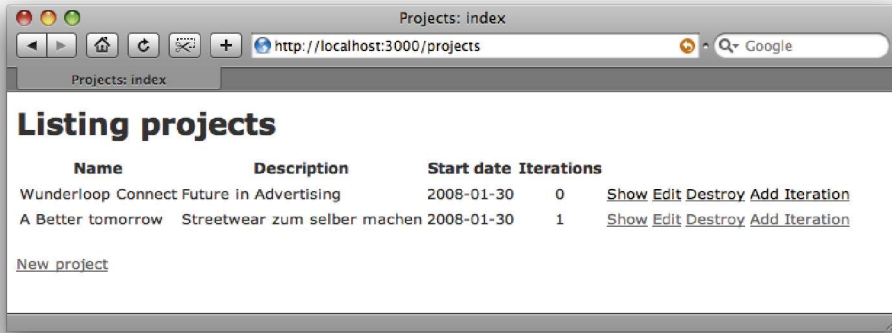


Abbildung 3.5: Projektübersicht inklusive Anzahl der Iterationen

3.7 Zwischenstand

Die erledigte Aufgabe hat uns das erste Mal in Kontakt mit der Rails-Programmierung gebracht. Wir haben Modellklassen um Assoziationen erweitert. Eine Iteration gehört zu genau einem Projekt (*belongs_to*), und ein Projekt besteht aus mehreren Iterationen (*has_many*).

Des Weiteren haben wir RHTML-Views kennengelernt und um einen zusätzlichen Link erweitert. Die von dem Link referenzierte Action *add_iteration* haben wir als neue öffentliche Methode der Klasse *ProjectsController* programmiert.

Zum Abschluss des Backlog-Items haben wir noch einen komplett neuen View für die Erfassung von Iterationen programmiert und dabei u.a. die Formular-Helper *form_tag*, *text_field*, *date_select* und *submit_tag* kennengelernt.

3.8 Iterationen anzeigen

Eine weitere wichtige Funktion ist die Anzeige bereits erfasster Iterationen. Ein guter Ausgangspunkt für diese Funktion ist der Show-View eines Projekts. Dieser View zeigt die Details erfasster Projekte an. Da wir Projekte in den vorangehenden Arbeitsschritten um Iterationen erweitert haben, müssen wir zunächst den Show-View um die Anzeige der zugehörigen Iterationen erweitern:

Listing 3.13: `app/views/projects/show.html.erb`

```
...
<table>
```

```

<tr>
  <th>Name</th>
  <th>Start</th>
  <th>End</th>
</tr>
<% for iteration in @project.iterations %>
<tr>
  <td><%=h iteration.name %></td>
  <td><%= iteration.start_date %></td>
  <td><%= iteration.end_date %></td>
  <td><%= link_to "Show", :action => "show_iteration",
                        :id => iteration %></td>
</tr>
<% end %>
</table>
...

```

Die Erweiterung besteht aus einer for-Schleife, die über die Liste der Iterationen des aktuell angezeigten Projekts iteriert. Für jede Iteration wird der Name sowie das Start- und Enddatum ausgegeben. Zusätzlich haben wir die Gelegenheit genutzt und jeder Iteration einen Link auf die Action *show_iteration* zugefügt. Abbildung 3.6 zeigt den um Iterationen erweiterten Show-View.

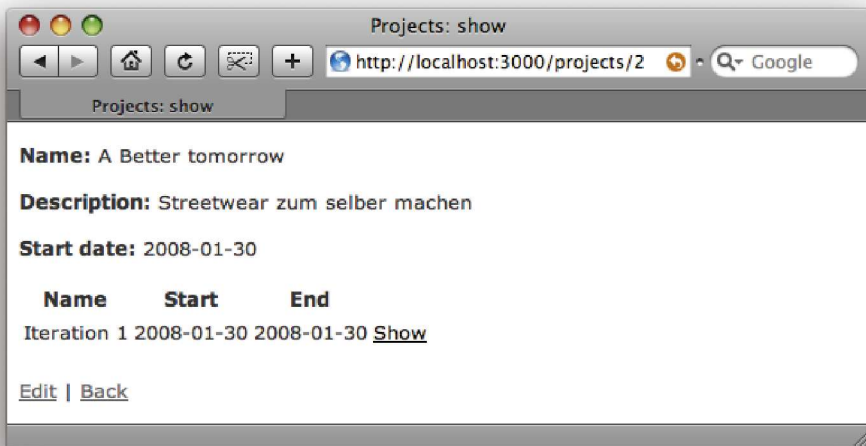


Abbildung 3.6: Ein Projekt mit Iterationen

Der Show-Link zeigt auf die Controller-Action *show_iteration*, die wie folgt implementiert ist:

Listing 3.14: `app/controllers/projects_controller.rb`

```
def show_iteration
```

```

    @iteration = Iteration.find(params[:id])
    render :template => "iterations/show"
  end

```

Die Action lädt die Iteration mit der als Parameter übergebenen ID und liefert anschließend den View zur Anzeige einer Iteration, den Sie folgendermaßen programmieren müssen:

Listing 3.15: `app/views/iterations/show.html.erb`

```

<% for column in Iteration.content_columns %>
<p>
  <b><%= column.human_name %></b>
  <%=h @iteration.send(column.name) %>
</p>
<% end %>
<%= link_to "Back", :action => "show",
               :id => @iteration.project %>

```

3.9 Iterationen bearbeiten und löschen

Bisher können wir Iterationen hinzufügen und anzeigen. Was noch fehlt, sind Funktionen zum Bearbeiten und Löschen von Iterationen. Als Erstes müssen wir dafür den Show-View für Projekte erweitern. Jede Iteration erhält zwei weitere Links, *edit* und *destroy*:

Listing 3.16: `app/views/projects/show.html.erb`

```

...
<% for iteration in @project.iterations %>
<tr>
  ...
  <td>
    <%= link_to "Show",      :action => "show_iteration",
                        :id => iteration %>
    <%= link_to "Edit",      :action => "edit_iteration",
                        :id => iteration %>
    <%= link_to "Destroy",  :action => "destroy_iteration",
                        :id => iteration %>
  </td>
</tr>
<% end %>
...

```

Der Edit-Link verweist auf die Action *edit_iteration*, die im *ProjectsController* wie folgt implementiert wird:

Listing 3.17: `app/controllers/projects_controller.rb`

```

def edit_iteration
  @iteration = Iteration.find(params[:id])

```

```
render :template => "iterations/edit"
end
```

Für die Bearbeitung von Iterationen verwenden wir denselben View wie für das Anlegen von neuen Iterationen, d.h. die Action liefert den View `app/views/iterations/edit.html.erb`.

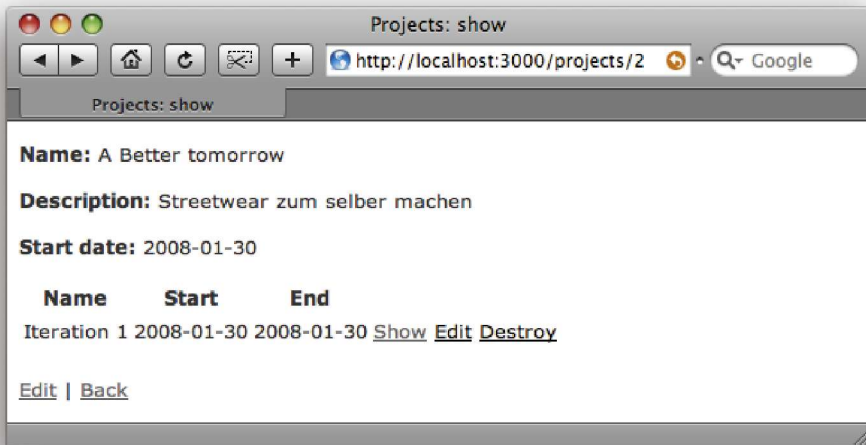


Abbildung 3.7: Neue Links: Edit und Destroy

Allerdings haben wir jetzt ein kleines Problem: Der Edit-View für Iterationen überträgt seine Daten an die von uns bereits implementierte Action `update_iteration`. Da diese Action ursprünglich für die Neuanlage von Iterationen programmiert wurde, erzeugt und speichert sie eine neue Iteration. In diesem Fall wollen wir aber eine vorhandene Iteration aktualisieren und speichern, d.h. wir müssen `update_iteration` so umbauen, dass die Action zwischen neuen und existierenden Iterationen unterscheidet.

Zur Erinnerung: Der Edit-View liefert in seinem Formular-Tag die ID der gerade bearbeiteten Iteration:

Listing 3.18: `app/views/iterations/edit.html.erb`

```
<% form_tag :action => "update_iteration",
           :id => @iteration do %>
  ...
```

Neue, d.h. noch nicht gespeicherte Iterationen unterscheiden sich von existierenden darin, dass die ID im ersten Fall `nil` ist und im zweiten Fall einen gültigen Wert besitzt. Diese Tatsache können wir in der Action `update_iteration` ausnutzen und so unterscheiden, ob der Benutzer eine neue oder eine vorhandene Iteration bearbeitet hat.

Listing 3.19: `app/controllers/projects_controller.rb`

```
def update_iteration
  if params[:id]
    @iteration = Iteration.find(params[:id])
  else
    @iteration = Iteration.new
  end

  if @iteration.update_attributes(params[:iteration])
    flash[:notice] = "Iteration was successfully updated."
    redirect_to(projects_url)
  else
    render :template => "iterations/edit"
  end
end
```

In Abhängigkeit davon, ob `params[:id]` einen gültigen Wert besitzt, laden wir entweder die existierende Iteration aus der Datenbank (*Iteration.find*) oder legen eine neue an (*Iteration.new*). Der sich an diese Fallunterscheidung anschließende Code ist dann für beide Fälle identisch: Die Attribute der Iteration werden basierend auf den Request-Parametern aktualisiert und gespeichert (*update_attributes*). Der Code kann noch etwas optimiert, d.h. mehr Ruby-like geschrieben werden:

Listing 3.20: `app/controllers/projects_controller.rb`

```
def update_iteration
  @iteration = Iteration.find_by_id(params[:id]) || Iteration.new
  ...
end
```

Statt der Methode *find* wird die Methode *find_by_id* verwendet, die ebenfalls die Iteration zur ID liefert. Im Falle eines nicht existierenden Datensatzes wirft sie aber keine Ausnahme, sondern liefert *nil* zurück. Das machen wir uns in Kombination mit der Oder-Verknüpfung zu Nutze. Existiert die Iteration, erhält *@iteration* die geladene Instanz. Existiert die Instanz nicht, wird der rechte Teil der Oder-Verknüpfung ausgeführt, der eine neue Instanz der Iteration im Speicher erzeugt.

Um das Backlog-Item abzuschließen, fehlt noch eine Action zum Löschen von Iterationen. Den Link dafür haben wir bereits dem Show-View für Projekte hinzugefügt. Der Link verweist auf die Action *destroy_iteration*, die im *ProjectsController* implementiert wird:

Listing 3.21: `app/controllers/projects_controller.rb`

```
def destroy_iteration
  iteration = Iteration.find(params[:id])
  project = iteration.project
  iteration.destroy
  redirect_to project
end
```

Beachten Sie, dass wir uns das zugehörige Projekt merken, bevor wir die Iteration löschen. Dies ist notwendig, damit wir nach dem Löschen auf die *show*-Action weiterleiten können, die als Parameter die Projekt-ID erwartet.

3.10 Tasks hinzufügen

Bisher können wir unsere Arbeit nur mit Hilfe von Projekten und Iterationen organisieren. Zur Erfassung der wirklichen Arbeit, d.h. der eigentlichen Aufgaben, steht bisher noch keine Funktion zur Verfügung. Das wollen wir ändern, indem wir unser System um eine Funktion zur Erfassung von Tasks erweitern. Als Erstes erzeugen wir eine entsprechende Modellklasse *Task*:

```
$ ruby script/generate model task name:string priority:integer \
  iteration_id:integer
...
```

Anschließend aktualisieren wir die Datenbank wieder per:

```
$ rake db:migrate
```

Beachten Sie bitte den Fremdschlüssel *iteration_id*, der die 1:N-Beziehung zwischen Iterationen und Tasks modelliert. Diese Beziehung benötigen wir zusätzlich auf Modellebene, d.h. die Klasse *Task* muss um eine *belongs_to*-Assoziation erweitert werden:

Listing 3.22: app/models/task.rb

```
class Task < ActiveRecord::Base
  belongs_to :iteration
end
```

Das Hinzufügen von Tasks soll genauso einfach sein wie das von Iterationen zu Projekten. Deshalb erweitern wir die Schleife über alle Iterationen eines Projekts um einen zusätzlichen Link *add_task*:

Listing 3.23: app/views/projects/show.html.erb

```
...
<% for iteration in @project.iterations %>
<tr>
  <td><%=h iteration.name %></td>
  ...
  <td>
    <%= link_to "Show",          :action => "show_iteration",
                                :id => iteration %>
    ...
    <%= link_to "Add Task", :action => "add_task",
                                :id => iteration %>
  </td>
  ...
</tr>
```

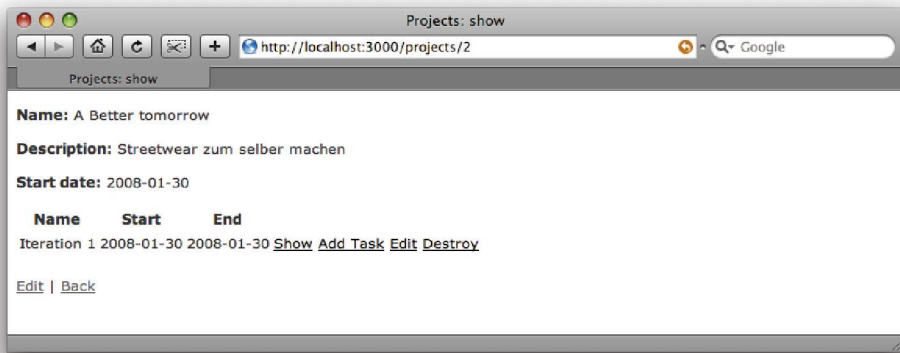



Abbildung 3.8: Der Show-View eines Projekts ermöglicht das Hinzufügen von Tasks

Abbildung 3.8 zeigt den erweiterten View *project/show.html.erb*.

Als Nächstes benötigen wir die Action *add_task*, die wir im Link schon verwenden, bisher jedoch noch nicht implementiert haben. Wir implementieren die Action im *ProjectsController*:

Listing 3.24: *app/controllers/projects_controller.rb*

```
def add_task
  iteration = Iteration.find(params[:id])
  @task = Task.new(:iteration => iteration)
  render :template => "tasks/edit"
end
```

Das Vorgehen ist nahezu identisch mit dem Hinzufügen von Iterationen zu Projekten. Die Action *add_task* liefert den View *app/views/tasks/edit.html.erb*, den wir wie folgt implementieren:

Listing 3.25: *app/views/tasks/edit.html.erb*

```
<h1>Editing Task</h1>
<%= error_messages_for :task %>
<% form_tag :action => "update_task", :id => @task do %>
  <p>Name: <%= text_field "task", "name" %></p>
  <p>Priority: <%= select(:task, :priority, [1, 2, 3]) %></p>
  <%= submit_tag "Update" %>
  <%= hidden_field "task", "iteration_id" %>
<% end -%>
<%= link_to "Back", :action => "show",
              :id => @task.iteration.project %>
```

Auch hier verwenden wir ein Hidden-Field, um die ID der zum Task gehörenden Iteration zurück an den Server zu übertragen. Als Submit-Action referenziert das Formular die Methode *update_task*. Wie wir beim Hinzufügen von Iterationen gelernt haben, wird eine Update-Action sowohl für das Erzeugen neuer als auch für

die Aktualisierung vorhandener Tasks benötigt, sodass wir die Action gleich entsprechend programmieren können:

Listing 3.26: `app/controllers/projects_controller.rb`

```
def update_task
  if params[:id]
    @task = Task.find(params[:id])
  else
    @task = Task.new
  end
  if @task.update_attributes(params[:task])
    flash[:notice] = "Task was successfully updated."
    redirect_to :action => "show_iteration", :id => @task.iteration
  else
    render :template => "tasks/edit"
  end
end
```

Zur Kontrolle, ob das Hinzufügen von Tasks funktioniert, erweitern wir den Show-View für Projekte um eine Anzeige der Taskanzahl pro Iteration:

Listing 3.27: `app/views/projects/show.html.erb`

```
<h2>Iterations</h2>
<table>
<tr>
  ...
  <th>End</th>
  <th>Tasks</th>
</tr>
<% for iteration in @project.iterations %>
<tr>
  ...
  <td><%= iteration.end_date %></td>
  <td><%= iteration.tasks.length %></td>
  ...
</tr>
</table>
...
```

Der View ruft die Methode `tasks` auf dem Objekt `iteration` auf, einer Instanz der Modellklasse `Iteration`. Damit diese Methode zur Laufzeit des Systems auch wirklich zur Verfügung steht, muss die Klasse `Iteration` um eine entsprechende `has_many`-Deklaration erweitert werden:

Listing 3.28: `app/models/iteration.rb`

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

3.11 Tasks anzeigen, bearbeiten und löschen

Genau wie Iterationen müssen auch einmal erfasste Tasks angezeigt, bearbeitet und gelöscht werden können. Wir denken, dass der Show-View für Iterationen ein guter Ausgangspunkt für diese Funktionen ist. Entsprechend erweitern wir diesen View um eine Liste von Tasks. Jeder Task wird dabei mit jeweils einem Link zum Anzeigen, Bearbeiten und Löschen ausgestattet:

Listing 3.29: `app/views/iterations/show.html.erb`

```
<% for column in Iteration.content_columns %>
<p>
  <b><%= column.human_name %>:</b>
  <%=h @iteration.send(column.name) %>
</p>
<% end %>
<h2>List of Tasks</h2>
<table>
<tr>
  <th>Name</th>
  <th>Priority</th>
</tr>
<% for task in @iteration.tasks %>
<tr>
  <td><%=h task.name %></td>
  <td><%= task.priority %></td>
  <td>
    <%= link_to "Show",      :action => "show_task",
                                :id => task %>
    <%= link_to "Edit",      :action => "edit_task",
                                :id => task %>
    <%= link_to "Destroy",   :action => "destroy_task",
                                :id => task %>
  </td>
</tr>
<% end -%>
</table>
...
```

Der View iteriert über die Taskliste der aktuell angezeigten Iteration und gibt für jeden Task dessen Namen, die Priorität und die erwähnten Links aus. Abbildung 3.9 zeigt den erweiterten View `iterations/show.html.erb`.

Der Show-Link verweist auf die Action `show_task`, die wir im `ProjectsController` implementieren:

Listing 3.30: `app/controllers/projects_controller.rb`

```
def show_task
  @task = Task.find(params[:id])
  render :template => "tasks/show"
end
```

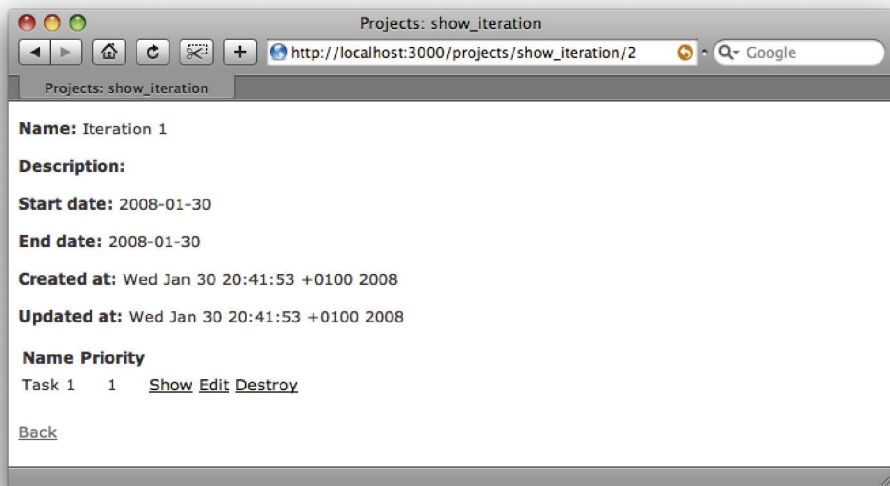


Abbildung 3.9: Ein neuer Show-View für Iterationen

Der von der Action gelieferte View `app/views/tasks/show.html.erb` sieht so aus:

```
<% for column in Task.content_columns %>
<p>
  <b><%= column.human_name %>:</b>
  <%= h @task.send(column.name) %>
</p>
<% end %>
<br>
<%= link_to "Back", :action => "show_iteration",
      :id => @task.iteration %>
```

Der Edit-Link referenziert die *ProjectsController*-Action *edit_task*:

Listing 3.31: `app/controllers/projects_controller.rb`

```
def edit_task
  @task = Task.find(params[:id])
  render :template => "tasks/edit"
end
```

Der zugehörige View `tasks/edit.html.erb` existiert bereits, da wir ihn schon im Rahmen der Neuanlage von Tasks erstellt haben.

Abschließend fehlt noch die Action für den Destroy-Link, der auf die *destroy*-Action des *ProjectsController* verlinkt:

Listing 3.32: `app/controllers/projects_controller.rb`

```
def destroy_task
  task = Task.find(params[:id])
```

```
iteration = task.iteration
task.destroy
redirect_to :action => "show_iteration", :id => iteration
end
```

3.12 Struktur in die Seiten bringen

Jetzt haben wir schon eine ganze Menge an Funktionalität entwickelt, und das System ist in der jetzigen Form rudimentär benutzbar. In diesem Abschnitt wollen wir ein wenig Struktur in die Seiten bekommen.

Häufig besteht eine Internetseite neben dem Inhaltsbereich aus einem Kopf, einem Seitenbereich links oder rechts und einer Fußzeile. Damit diese Elemente nicht in jeder Seite neu implementiert werden müssen, bietet Rails das einfache Konzept des *Layouts* (vgl. Abschnitt 6.4).

Layouts sind RHTML-Seiten, die die eigentliche Inhaltsseite umschließen. Für die Einbettung der Inhaltsseite steht in der Layout-Seite der Aufruf *yield* zur Verfügung. Genau an der Stelle im Layout, wo Sie diesen Aufruf benutzen, wird die darzustellende Seite eingebettet. Um sie herum können Sie nach Belieben andere Elemente, wie z.B. Navigation, News, Kopf- oder Fußzeile, einfügen:

```
<html>
<head>
...
</head>
<body>
...
<%= yield %>
...
</body>
</html>
```

Der Scaffold-Generator erzeugt für jeden Controller ein Standard-Layout im Verzeichnis *app/views/layouts*. In diesem Verzeichnis befindet sich also auch eine Datei *projects.html.erb*, die der Generator für unseren *ProjectsController* erzeugt hat. Auch hier profitieren wir von der Rails-Konvention, dass eine dem Controller-Namen entsprechende Layout-Datei automatisch als Standardlayout für diesen Controller verwendet wird.

Da wir das Layout aber nicht nur für die Views zum *ProjectsController* nutzen möchten, sondern für die gesamte Anwendung, benennen wir die Datei *projects.html.erb* in *application.html.erb* um. Dieses Layout wird von allen Controllern verwendet, sofern der Controller kein eigenes Layout besitzt, und in der Regel benötigen Sie nur ein Layout. Löschen Sie ggf. alle Controller-spezifischen Layout-Dateien aus *app/views/layouts/*.

Alles, was wir jetzt noch tun müssen, ist, unsere gestalterischen Fähigkeiten spielen zu lassen und entsprechend schöne HTML-Elemente und Stylesheets in diese Datei einzubauen. Ein erster Schritt wäre, ein Logo in die Titelleiste einzufügen. Des-

halb haben wir ein entsprechendes Logo erzeugt und in das Layout *app/views/layouts/projects.html.erb* eingefügt:

Listing 3.33: *app/views/layouts/application.html.erb*

```
...  
<div id="logo_b-simple">  
    
</div>  
...
```

Die Logo-Datei *logo.jpg* muss dafür im Verzeichnis *public/images* vorhanden sein. Abbildung 3.10 zeigt die Seite mit dem neuen Logo. Je nach Anforderung, Lust und Laune sind weitere Schritte zu einer ansprechenden Seite möglich.

Neben dem Layout-Konzept bietet Rails über so genannte *Partials* weitere Möglichkeiten, die Seiten in kleinere Elemente zu zerlegen und diese an verschiedenen Stellen wieder zu verwenden. Wir werden darauf in Abschnitt 6.5 eingehen.

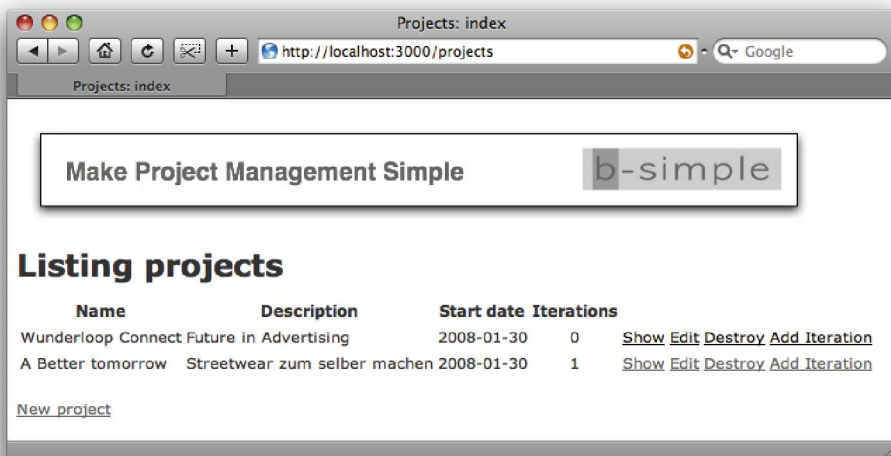


Abbildung 3.10: OnTrack-Seite mit Logo

3.13 Validierung

Die Validierung von Benutzereingaben macht eine Applikation wesentlich robuster und steht deshalb als nächste Aufgabe in unserem Backlog. Wir werden uns mit dem Thema Validierung ausführlich in Abschnitt 4.14 befassen und Ihnen im Folgenden wiederum einen ersten Eindruck liefern. Lassen Sie uns dazu einige Punkte sammeln bezüglich dessen, was es zu validieren gilt:

- Projekte müssen einen eindeutigen Namen haben.

- Iterationen müssen einen eindeutigen Namen haben.
- Tasks müssen einen Namen haben.
- Das Enddatum einer Iteration muss größer als das Startdatum sein.

Validierung findet in Rails auf Modellebene statt. Die einfachste Möglichkeit der Validierung ist die Erweiterung der Modellklasse um Validierungs-Deklarationen. Dies sind Klassenmethoden, die in die Klassendefinition eines Modells eingefügt werden. Wir beginnen mit der ersten Validierungsanforderung und erweitern die Klasse *Project* um die Validierung des Projektnamens. Hierfür verwenden wir die Methode *validates_presence_of*, die sicherstellt, dass das angegebene Attribut nicht leer ist:

Listing 3.34: app/models/project.rb

```
class Project < ActiveRecord::Base
  has_many :iterations, :dependent => :destroy
  validates_presence_of :name
  ...
end
```

Rails führt die Validierung vor jedem *save*-Aufruf des Modells durch. Schlägt dabei eine Validierung fehl, fügt Rails der Fehlerliste eines Modells *errors* einen neuen Eintrag hinzu und bricht den Speichervorgang ab. Die Methode *save* liefert einen Booleschen Wert, der anzeigt, ob die Validierung und damit das Speichern erfolgreich war. Diesen Rückgabewert werten wir bereits in der *ProjectsController*-Action *create* aus, die den New-View eines Projekts wiederholt öffnet, wenn die Validierung fehlschlägt:

Listing 3.35: app/controllers/projects_controller.rb

```
def create
  @project = Project.new(params[:project])

  respond_to do |format|
    if @project.save
      flash[:notice] = "Project was successfully created."
      format.html { redirect_to(@project) }
      format.xml { render :xml => @project,
        :status => :created, :location => @project }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @project.errors,
        :status => :unprocessable_entity }
    end
  end
end
```

Damit der Benutzer weiß, weshalb das Speichern fehlschlägt, müssen wir ihm die Liste dieser Fehlermeldungen anzeigen. Der Code dafür ist einfach und bereits (dank Scaffolding) in den Views *views/projects/new.html.erb* und *views/projects/edit.html.erb* enthalten:

Listing 3.36: `app/views/projects/new.html.erb`

```
<%= error_messages_for :project %>
...
```

Der View verwendet den Formular-Helper *error_messages_for*, der einen String mit den Fehlermeldungen des übergebenen Objekts zurückliefert. Sie müssen also nichts weiter tun, als das Modell um Aufrufe der benötigten Validierungsmethoden zu erweitern. Das Ergebnis einer fehlschlagenden Namensvalidierung sehen Sie in Abbildung 3.11. Neben Anzeige der Fehlerliste markiert der View zusätzlich die als fehlerhaft validierten Felder mit einem roten Rahmen.

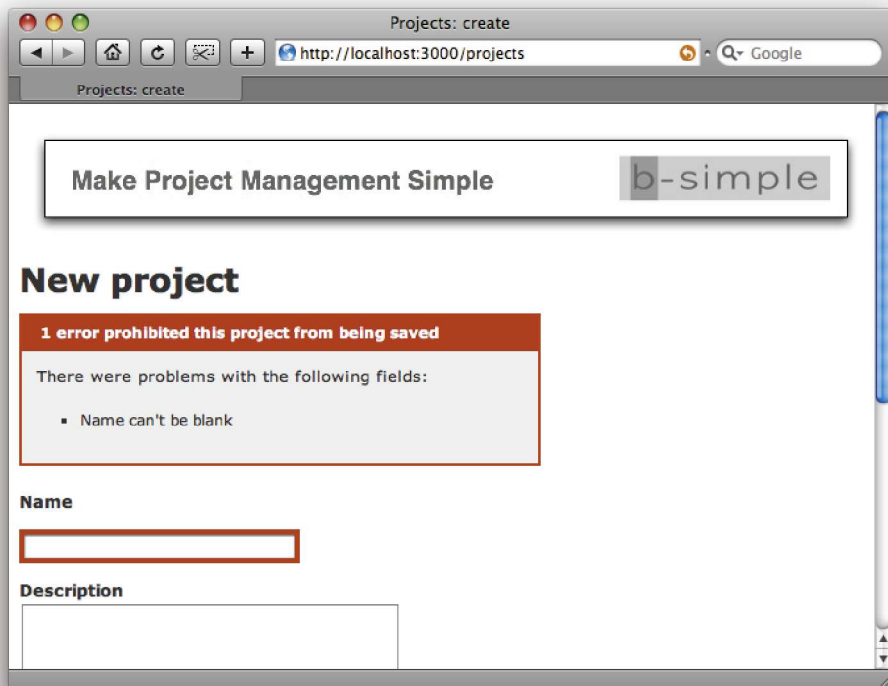


Abbildung 3.11: Projekte ohne Namen sind nicht erlaubt.

Rails verwendet hier eine Standardfehlermeldung in Englisch. In Kapitel 8 zeigen wir Ihnen, wie Sie Ihre Anwendung internationalisieren bzw. lokalisieren und damit auch Fehlermeldungen z.B. in Deutsch anzeigen können.

Als Nächstes gilt es, die Eindeutigkeit von Projektnamen sicherzustellen. Hierfür steht die Methode *validates_uniqueness_of* zur Verfügung, die wir zusätzlich in die Klasse *Project* einbauen:

Listing 3.37: `app/models/project.rb`

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



```
validates_presence_of :name  
validates_uniqueness_of :name  
...  
end
```

Wenn Sie jetzt einen Projektnamen ein zweites Mal vergeben, weist Sie die Anwendung auf diesen Fehler hin (siehe Abbildung 3.12).

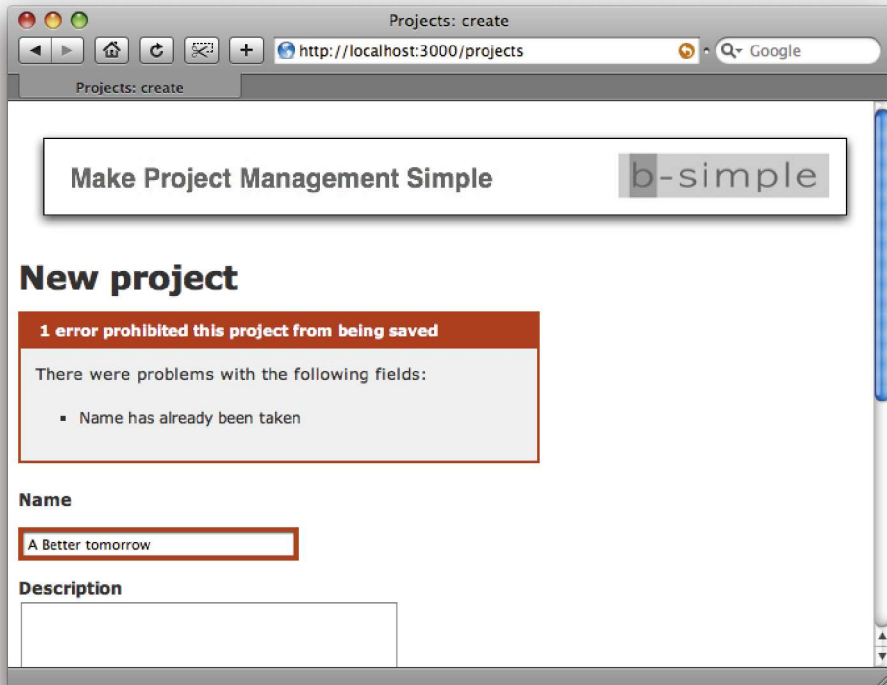


Abbildung 3.12: Projektnamen müssen eindeutig sein.

Zum Abschluss wollen wir Ihnen noch eine andere Art der Validierung erklären, die wir für die Überprüfung der Start- und Endtermine von Iterationen benötigen. Wir wollen sicherstellen, dass das Enddatum einer Iteration nach deren Startdatum liegt. In diesem Fall haben wir es mit zwei Attributen zu tun, die nur zusammen validiert werden können. Für diese Anforderung ist es sinnvoll, die *validate*-Methode der Klasse *ActiveRecord::Base* zu überschreiben, die Rails vor jedem Speichern des Modells aufruft.

In der Methode *validate* haben wir Zugriff auf die aktuellen Attributwerte der Iteration und können prüfen, ob das Enddatum größer als das Startdatum ist. Schlägt diese Überprüfung fehl, wird die Fehlerliste *errors* um einen weiteren Eintrag ergänzt:

Listing 3.38: app/models/iteration.rb

```
class Iteration < ActiveRecord::Base
  ...
  def validate
    if end_date <= start_date
      errors.add(:end_date,
        "Das Enddatum muss größer als das Startdatum sein")
    end
  end
end
```

Die Hash *errors* steht jeder Modellklasse über *ActiveRecord::Base* zur Verfügung. Rails prüft im Anschluss an den *validate*-Aufruf deren Inhalt. Enthält die Hash mindestens einen Fehler, wird das Speichern abgebrochen, und *save* liefert *false*. Wie in Listing 3.19 zu sehen, wird entsprechend dem Rückgabewert verzweigt. Damit die Fehlermeldungen angezeigt und die entsprechenden Felder rot umrandet werden, muss der Aufruf *error_messages_for :iteration* in den Edit-View für Iterationen eingebaut werden.

3.14 Benutzerverwaltung

Unsere Anwendung soll die Arbeit von Teams unterstützen und benötigt deshalb eine Benutzerverwaltung. Wir haben uns entschieden, im ersten Schritt eine einfache, Scaffold-basierte Benutzerverwaltung zu erstellen, auf die wir im nächsten Schritt die Login-Funktionalität aufbauen können.

Benutzer werden durch das Domain-Objekt *Person* modelliert, für das wir ein Modell sowie einen zuständigen Controller mit Hilfe des Scaffold-Generators erzeugen:

```
$ ruby script/generate scaffold person username:string \
  password:string firstname:string surname:string
```

Zum Speichern von Benutzern verwenden wir die Tabelle *people*.¹² Das folgende Migrationsskript enthält das entsprechende Schema und wurde von uns um die Erstellung eines Testbenutzers erweitert:

Listing 3.39: db/migrate/004.create_people.rb

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.string :username
      t.string :password
      t.string :firstname
      t.string :surname

      t.timestamps
    end
  end
end
```

¹²Rails kennt einige spezielle Pluralisierungsregeln der englischen Sprache. Mehr zu diesem Thema finden Sie in Abschnitt 4.1.4.

```
end

Person.create(:username => "ontrack", :password => "ontrack",
              :firstname => "Peter", :surname => "Muster" )
end

def self.down
  drop_table :people
end
end
```

Und wie bekannt, erfolgt die Aktualisierung der Datenbank durch den Aufruf:

```
$ rake db:migrate
```

Im Grunde genommen war das auch schon alles, was wir für eine erste rudimentäre Benutzerverwaltung tun müssen. Geben Sie die URL <http://localhost:3000/people> ein, und erfassen Sie einige Benutzer (siehe Abbildung 3.13).

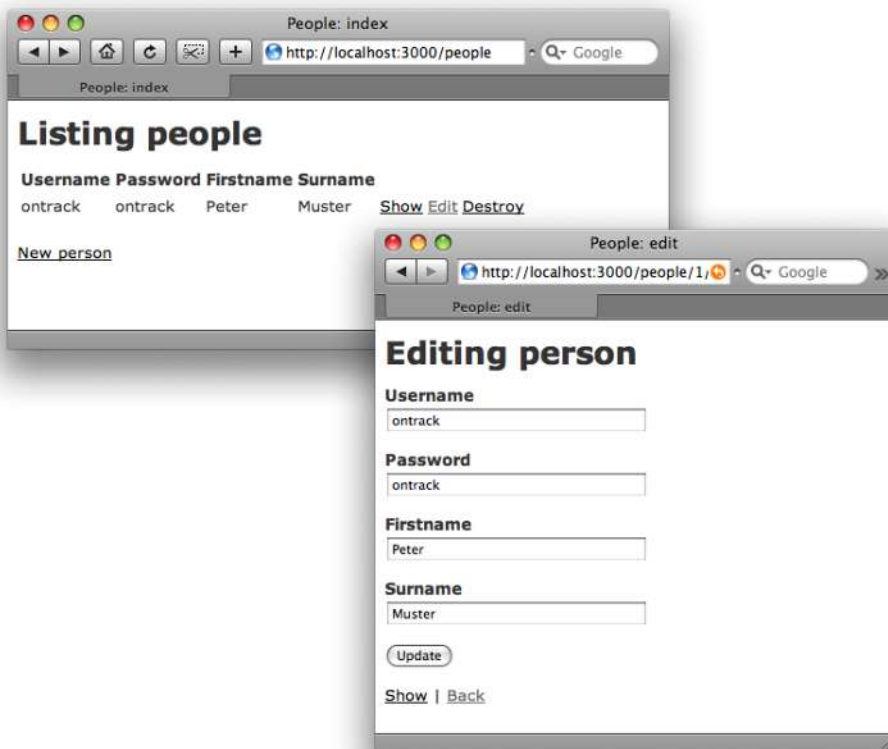


Abbildung 3.13: Die OnTrack-Benutzerverwaltung

Zwei Dinge fallen auf: Rails erzeugt im Edit-View ein spezielles Passwort-Feld. Weniger glücklich ist die Tatsache, dass Rails das Passwort im List-View im Klartext ausgibt. Sie können das Problem beheben, indem Sie die entsprechende Tabellenspalte aus dem View löschen.

3.15 Login

Spannender als die eigentliche Benutzerverwaltung ist die Programmierung des Login-Mechanismus.¹³

Das erste für die Benutzerverwaltung benötigte neue Konzept sind die so genannten *Filter* (vgl. Abschnitt 5.10). Ein Filter installiert eine Methode, die Rails vor bzw. nach der Ausführung einer Action automatisch aufruft. Damit soll erreicht werden, dass Rails vor der Ausführung einer Action automatisch prüft, ob der Benutzer beim System angemeldet ist. Die einfachste Möglichkeit, dies zu realisieren, ist die Installation eines Before-Filters in der zentralen Controllerklasse *ApplicationController*, von der alle Controller unserer Anwendung erben:

Listing 3.40: app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  ...
  before_filter :authenticate

  protected
  def authenticate
    redirect_to :controller => "authentication",
               :action => "login"
  end
end
```

Ruby: Sichtbarkeit von Methoden

Die Sichtbarkeit von Methoden wird in Ruby durch die Schlüsselwörter *public*, *protected* und *private* definiert. Sie führen einen Bereich ein, in dem alle enthaltenen Methoden so lange die gleiche Sichtbarkeit haben (z.B. *private*), bis diese durch ein anderes Schlüsselwort (z.B. *public*) beendet wird. Per Default sind alle Methoden einer Klasse von außen sichtbar, d.h. *public*.

Der Filter wird durch den Aufruf der Methode *before_filter* installiert, die die aufzurufende Methode als Symbol erhält. Die Installation des Filters in unserer zentralen Controller-Basisklasse bewirkt, dass die Methode *authenticate* vor Ausführung jeder Controller-Action unserer Anwendung aufgerufen wird. Die Methode wird in ihrer Sichtbarkeit durch *protected* eingeschränkt, damit sie nicht von außen aufzurufen ist (siehe Kasten *Sichtbarkeit von Methoden*).

¹³Für die Login-Funktionalität gibt es bereits Plugin (z.B. *restful_authentication*), die wir hier aber nicht verwenden, weil wir die notwendigen Schritte explizit zeigen möchten.

Um zu testen, ob das Ganze funktioniert, haben wir in der ersten Version der Methode eine einfache Weiterleitung auf die *login*-Action des neuen *AuthenticationController* programmiert. Die eigentliche Prüfung haben wir erst mal weggelassen:

Listing 3.41: `app/controllers/authentication_controller.rb`

```
$ ruby script/generate controller authentication

class AuthenticationController < ApplicationController
  skip_filter :authenticate
end
```

Wir haben den generierten *AuthenticationController* um den Aufruf der Methode *skip_filter* erweitert. Die Methode bewirkt, dass der im *ApplicationController* zentral installierte Before-Filter *authenticate* für den *AuthenticationController* nicht ausgeführt wird. Ohne diese Filter-Unterdrückung würde jeder Login-Versuch in einen niemals endenden Kreislauf münden. Logisch, oder?

Des Weiteren benötigen wir einen neuen View *login*, der Felder zur Eingabe von Benutzernamen und Passwort enthält:

Listing 3.42: `app/views/authentication/login.html.erb`

```
<% form_tag :action => "sign_on" do %>
<table>
  <tr><td>Username:</td>
    <td><%= text_field_tag :username %>
  </tr><tr>
    <td>Password:</td>
    <td><%= password_field_tag :password %></td>
  </tr>
</table>
<%= submit_tag "Login" %>
<% end -%>
```

Die in dem View verwendeten Helper *text_field_tag* und *password_field_tag* unterscheiden sich von den bisher verwendeten Helpers darin, dass sie keinerlei Modellbezug besitzen.

Egal, welche URL Sie jetzt eingeben, der Filter sorgt immer dafür, dass Sie auf die *login*-Action des *AuthenticationController* umgeleitet werden, die Sie auffordert, Benutzernamen und Passwort einzugeben (siehe Abbildung 3.14).

Jetzt haben wir zwar unser gesamtes System lahmgelegt, konnten aber zumindest testen, ob der installierte Filter greift. Um unser System wiederzubeleben, müssen zwei Dinge getan werden: Erstens muss die im Login-View angegebene Action *sign_on* implementiert und zweitens die Methode *authenticate* aus dem *ApplicationController* um eine Überprüfung, ob ein Benutzer angemeldet ist, erweitert werden.

Bevor wir aber mit der Programmierung beginnen, müssen wir ein weiteres neues Konzept einführen: *Sessions*. Eine Session ist ein Request-übergreifender Datenspeicher, der in jedem Request und somit in jeder Action zur Verfügung steht (vgl. Ab-

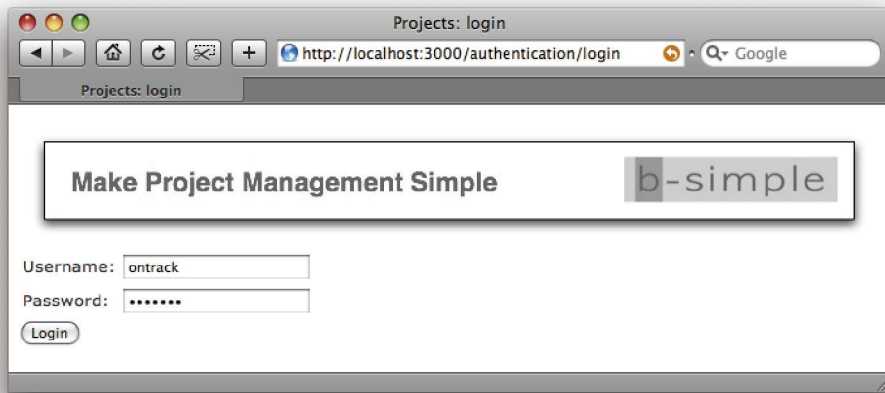


Abbildung 3.14: Die Login-Seite

schnitt 5.8). Wir nutzen die Session, um uns den erfolgreich angemeldeten Benutzer zu merken.

Dabei speichern wir nicht die Instanz der Person, sondern nur dessen ID. Benötigen wir die Instanz, laden wir diese über die ID aus der Datenbank. Dadurch sind die Daten zum aktuellen Benutzer immer aktuell. Andernfalls müssen Sie sicherstellen, dass Änderungen im Speicher mit der Datenbank synchronisiert werden und umgekehrt. Das mag für das Beispiel des Benutzers noch übertrieben wirken, aber sobald Sie sich selbst um die Synchronisation kümmern müssen, kann Ihnen das eine Menge Probleme einbringen. Speichern Sie nur die ID in der Session.

Listing 3.43: `app/controllers/authentication_controller.rb`

```
class AuthenticationController < ApplicationController
  skip_filter :authenticate

  def sign_on
    person = Person.find(:first, :conditions =>
      ["username = ? AND password = ?",
        params[:username], params[:password]])
    if person
      session[:person] = person.id
      redirect_to projects_url
    else
      render :action => "login"
    end
  end
end
```

Der *ApplicationController* wird so erweitert, dass nur dann auf die Login-Action umgeleitet wird, wenn sich noch kein Person-Objekt in der Session befindet, d.h. der Benutzer nicht angemeldet ist:

Listing 3.44: app/controllers/application.rb

```
def authenticate
  if Person.find_by_id(session[:person]).nil?
    redirect_to :controller => "authentication", :action => "login"
  end
end
```

Wenn Sie wollen, können Sie das System noch um eine Abmelde-Action erweitern, bei der die Session per *reset_session* zurückgesetzt wird. Vielleicht zeigen Sie über das Layout *application.html.erb* den angemeldeten Benutzer an und bieten einen Link *Logout*.

Listing 3.45: app/controllers/authentication_controller.rb

```
def logout
  reset_session
  redirect_to :action => "login"
end
```

3.16 Tasks zuweisen

Unser System dient nicht nur der Erfassung und Bearbeitung von Tasks. Irgendwann soll die Arbeit auch richtig losgehen, d.h. Projektmitglieder müssen sich für einzelne Tasks verantwortlich erklären. Um diese Verantwortlichkeiten im System pflegen zu können, werden wir zunächst das Datenmodell um eine 1:N-Beziehung zwischen Tasks und Personen erweitern.

Dazu erweitern wir die Tabelle *tasks* um einen Fremdschlüssel *person_id*, indem wir ein neues Migrationsskript ohne Modell erzeugen:

```
$ ruby script/generate migration add_person_id_to_tasks
exists db/migrate
create db/migrate/005_add_person_id_to_tasks.rb
```

In diesem Skript erweitern wir die Tabelle *tasks* um das Attribut *person_id*:

Listing 3.46: db/migrate/005_add_person_id_to_tasks.rb

```
class AddPersonIdToTasks < ActiveRecord::Migration
  def self.up
    add_column :tasks, :person_id, :integer
  end

  def self.down
    remove_column :tasks, :person_id
  end
end
```

Hier verwenden wir die Methoden *add_column* und *remove_column* zum Hinzufügen und Löschen von Attributen einer Tabelle. Der erste Parameter gibt dabei

die Datenbanktabelle an und der zweite den Attributnamen. Es folgt der obligatorische Aufruf von:

```
$ rake db:migrate
```

Als Nächstes wird die Assoziation auf Modellebene modelliert, indem die Klasse *Task* um die entsprechende *belongs_to*-Deklaration erweitert wird:

Listing 3.47: `app/models/task.rb`

```
class Task < ActiveRecord::Base
  belongs_to :iteration
  belongs_to :person
end
```

Abschließend muss noch der Edit-View für Tasks um eine Zuordnungsmöglichkeit der verantwortlichen Person erweitert werden:

Listing 3.48: `app/views/tasks/edit.html.erb`

```
...
<p>Priority: <%= select(:task, :priority, [1, 2, 3]) %></p>
<p>Responsibility:
  <%= collection_select(:task, :person_id,
                        Person.find(:all, :order => "surname"),
                        :id, :surname) %>
</p>
...
```

Wir verwenden dafür den Formular-Helper *collection_select*. Die Methode erzeugt eine Auswahlbox mit Benutzernamen. Die ersten beiden Parameter *:task* und *:person_id* geben an, in welchem Request-Parameter die Auswahl an den Server übertragen wird (hier *task[person_id]*).

Der dritte Parameter ist die Liste der darzustellenden Personen. Die beiden letzten Parameter *:id* und *:surname* geben an, welche Methoden der Objekte in der Liste aufgerufen werden, um für jeden Listeneintrag die ID und den darzustellenden Wert zu ermitteln. Das Ergebnis der View-Erweiterung ist in Abbildung 3.15 dargestellt.

3.17 Endstand und Ausblick

Wir haben in diesem Kapitel eine sehr einfache Projektmanagement-Software entwickelt und dabei einige zentrale Rails-Komponenten kennengelernt:

- Migration
- Scaffolding
- Active Record (Modelle, Modell-Assoziationen, Validierung)
- Action Controller (Actions, Sessions)
- Action View (View, Templates, Formular-Helper, Layouting)

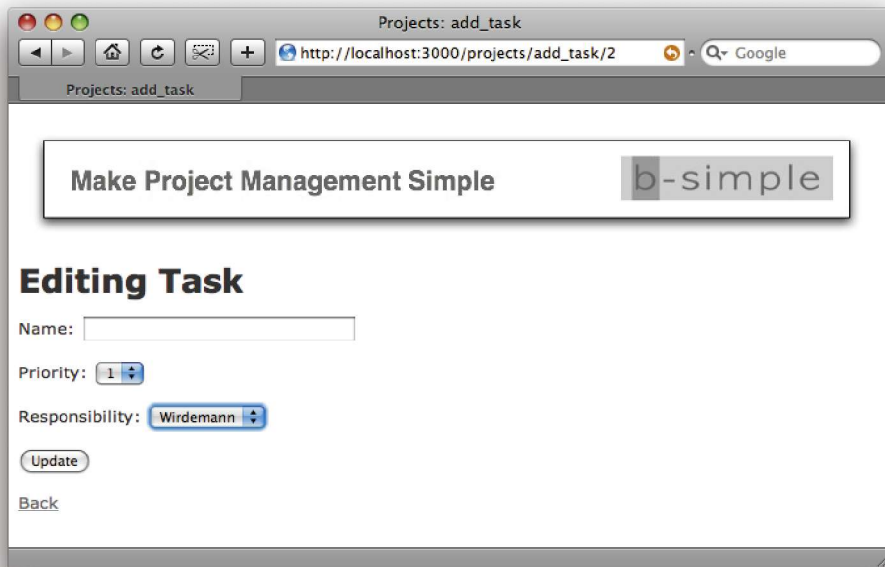


Abbildung 3.15: Zuordnung von Tasks

Über die Darstellung der einzelnen Rails-Komponenten hinaus war es uns in diesem Kapitel wichtig, einige der zentralen Rails-Philosophien zu beschreiben:

- Unmittelbares Feedback auf Änderungen
- Konvention über Konfiguration
- DRY (siehe Abschnitt 2.1)
- Wenig Code

Das in diesem Kapitel entwickelte System ist natürlich lange noch nicht vollständig. Es fehlt z.B. eine Möglichkeit, Benutzern Projekte zuzuweisen. Eine gute Möglichkeit für Sie, das System zu erweitern und sich mit Rails vertraut zu machen.

Alle Actions sind in einem Controller *ProjectsController* gelandet. Besser ist es, für die Actions zur Bearbeitung der Iterationen und Tasks jeweils einen eigenen Controller bereitzustellen. Die Verwendung eines Controllers war im Rahmen dieses Kapitels aber die einfachste Möglichkeit, die schrittweise Entwicklung von Modellen, Views und zugehörigen Actions zu beschreiben.

In den folgenden Abschnitten werden wir in die Details des Rails-Frameworks einsteigen und die hier teilweise nur oberflächlich angerissenen Themen ausführlich beschreiben.

Kapitel 4

Active Record

Das *Active Record*-Framework ist eines der drei Sub-Frameworks von Rails. Das Framework stellt die Verbindung zwischen Domain-Objekten und Datenbank her und ermöglicht die komfortable Speicherung von Objekten in der zugrunde liegenden Datenbank.

Active Record-Objekte kapseln Daten und Geschäftslogik. Sie beziehen ihre Attribute direkt aus der zugehörigen Datenbanktabelle. Änderungen am Datenmodell haben unmittelbare Auswirkungen auf das Domain-Modell. Duplizierung, d.h. redundante Wiederholung von Informationen, findet nicht statt.

4.1 Active Record-Klassen

Active Record basiert auf dem gleichnamigen Pattern (siehe Kasten *Active Record – das Pattern*) zur Abbildung von Objektmodellen auf relationale Datenbanken. Eine Active Record-Klasse repräsentiert dabei eine Datenbanktabelle und ein Objekt dieser Klasse eine Datenzeile in dieser Tabelle.

Active Record-Klassen erben von *ActiveRecord::Base*:

```
class User < ActiveRecord::Base
end
```

Die Verbindung zwischen einer Active Record-Klasse und ihrer zugehörigen Datenbanktabelle stellt Rails über den Namen der Klasse her, indem als Tabellename die pluralisierte Form des Klassennamens verwendet wird. Objekte der Klasse *User* werden in der Tabelle *users* gespeichert:

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :firstname, :lastname
    end
  end
end
```

```

def self.down
  drop_table :users
end
end

```

Active Record – das Pattern

Dem Active Record-Framework liegt das gleichnamige Pattern *Active Record* zugrunde, das von Martin Fowler in [7] beschrieben wurde.

Zentrale Idee von Active Record ist die Verwendung von Klassen zur Repräsentation einer Datenbanktabelle. Eine Active Record-Klasse korrespondiert dabei mit genau einer Datenbanktabelle. Eine Instanz einer Active Record-Klasse repräsentiert genau eine Datenzeile in dieser Tabelle. Die Felder einer Active Record-Klasse müssen 1:1 mit den Feldern der zugehörigen Tabelle korrespondieren. Während in Java dafür entsprechende Attribute mit Gettern und Settern programmiert werden (DRY-Verletzung, siehe auch Abschnitt 2.1), werden die Attribute von Rails Active Records ausschließlich in der Tabelle definiert (DRY-Einhaltung, siehe auch Abschnitt 2.1). Dank Ruby ist es möglich, Active Record-Klassen zur Laufzeit um Attribute und Zugriffsmethoden für die zugehörigen Tabellenfelder zu erweitern.

Active Record-Klassen spezifizieren ihre Attribute nicht direkt, sondern beziehen sie aus der Tabellendefinition der zugehörigen Datenbanktabelle.¹ Für jedes Feld der zugehörigen Tabelle erzeugt Active Record eine Getter- und eine Setter-Methode. Beispielsweise wird die Klasse *User* für das Tabellenfeld *lastname* um die beiden Methoden *lastname* und *lastname=* erweitert:

```

user = User.create(:lastname => "Wirdemann")
puts "Lastname: #{user.lastname}"      # der Getter
user.lastname = "Baustert"             # der Setter

```

Darüber hinaus erzeugt Active Record für jedes Tabellenfeld eine Instanzvariable. Um innerhalb der Modellklasse zwischen Instanzvariable und Getter- und Setter-Methoden zu unterscheiden, muss der Instanzvariablen das Schlüsselwort *self* vorangestellt werden:

```

class User < ActiveRecord::Base
  def fullname
    firstname + " " + lastname # Getter
    self.lastname              # Instanzvariable
  end
end

```

¹ Selbstverständlich kann eine Active Record-Klasse neben den aus der Tabelle bezogenen Attributen weitere Attribute definieren.

Überschreiben von Gettern und Settern

Dynamisch erzeugte Getter und Setter können überschrieben werden. Um dabei rekursive Aufrufe zu vermeiden, darf der jeweilige Getter bzw. Setter in der überschriebenen Methode nicht verwendet werden. Stattdessen stehen die Methoden *write_attribute* und *read_attribute* bzw. die direkte Nutzung der Hash *self* zur Verfügung:

```
class User < ActiveRecord::Base
  def lastname=(name)
    write_attribute(:lastname, name) # entweder so,
    self[:lastname] = name           # oder so
  end

  def lastname
    read_attribute(:lastname)        # entweder so,
    self[:lastname]                  # oder die Hash
  end
end
```

4.1.1 Mehr über Modellattribute

Tabellenfelder definieren die Attribute von Modellklassen. Beim Lesen und Schreiben von Modellinstanzen bildet Active Record die Attribute bzw. Felder auf den korrespondierenden Typ des jeweils anderen Systems ab. Die Abbildung von „einfachen“ Datentypen², wie Strings oder Fixnums, führt Active Record automatisch durch.

Komplexere Datentypen wie Arrays, Hashes oder andere serialisierbare Objekte³ werden in Tabellenfeldern vom Typ *Text* gespeichert. Als Beispiel erweitern wir die Tabelle *users* um das Feld *preferences* zum Speichern von speziellen Benutzereinstellungen:

```
class AddPreferencesToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :preferences, :text
  end

  def self.down
    remove_column :users, :preferences
  end
end
```

In der Modellklasse *User* werden Benutzereinstellungen in einer Hash gespeichert. Für die Konvertierung der Hash in ein Textfeld und umgekehrt muss das Attribut in der Modellklasse explizit als serialisierbar deklariert werden. Diesem Zweck dient die Methode *serialize*:

```
class User < ActiveRecord::Base
```

² Nicht zu verwechseln mit primitiven Datentypen, die es in Ruby nicht gibt.

³ Im Prinzip alle Objekte, die sich mit Hilfe von YAML serialisieren lassen.

```

    serialize :preferences
  end

  preferences = {"farbe" => "gruen", "anzahl" => 11}
  user = User.create(:preferences => preferences)
  puts user.preferences.inspect

$ {"anzahl"=>11, "farbe"=>"gruen"}
```

Active Record führt das Typ-Mapping automatisch durch. Wenn *preferences* an Stelle einer Hash mit einem Array initialisiert wird, dann wird beim nächsten Laden des Objekts *user* ein Array instanziiert:

```

user = User.create(:preferences => %w(blau rot))
puts user.preferences.inspect

$ ["blau", "rot"]
```

Ein optionaler Typ-Parameter von *serialize* schränkt die möglichen Typen des zu serialisierenden Objekts auf genau einen Typ ein:

```

class User < ActiveRecord::Base
  serialize :preferences, Hash
end
```

In diesem Fall wirft Active Record beim Zugriff auf *preferences* eine *SerializationTypeMismatch*-Exception, wenn das Attribut *preferences* zuvor als Array gespeichert wurde:

```

user = User.create(:preferences => %w(blau rot))

$ ActiveRecord::SerializationTypeMismatch: preferences was
  supposed to be a Hash, but was a Array
```

4.1.2 Mehr über Primärschlüssel

Jede Active Record-Tabelle benötigt einen Primärschlüssel. Standardmäßig nimmt Active Record hierfür den Namen *id* an. Hält man sich an diese Konvention, dann wird die Verwaltung des Primärschlüssels vollständig von Active Record übernommen. Für neue Datensätze wird deren initialer ID-Wert von der Datenbank erzeugt:

```

user = User.new(:firstname => "Ralf")
user.save
puts user.id

$ 1
```

Der Primärschlüssel eines Modells kann über die Methode *set_primary_key* explizit angegeben werden. Das folgende Beispiel erweitert die Tabelle *users* um ein Feld *mobile_no* und macht dieses Feld zum Primärschlüssel des *User*-Modells:

```

class AddMobileNoToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :mobile_no, :integer
  end

  def self.down
    remove_column :users, :mobile_no
  end
end

class User < ActiveRecord::Base
  set_primary_key :mobile_no
end

```

Für den Zugriff auf den Primärschlüssel einer Modellklasse verwendet Active Record intern weiterhin die Getter-Methode *id*, d.h. diese Methode steht Ihnen weiterhin zur Verfügung und liefert demzufolge den gleichen Wert wie die Getter-Methode des explizit spezifizierten Primärschlüssels:

```

user = User.create(:firstname => "Ralf")
puts "Mobile: #{user.mobile_no}"
puts "Id      : #{user.id}"

$ Mobile: 12
  Id      : 12

```

4.1.3 Zusammengesetzte Primärschlüssel

Viele Tabellen in Legacy-Datenbanken verfügen über keinen technischen Primärschlüssel und verwenden stattdessen aus Domain-Daten zusammengesetzte Schlüssel. Rails unterstützt keinen zusammengesetzten Primärschlüssel. Vielleicht kann Ihnen das Plugin unter <http://compositekeys.rubyforge.org/> helfen.

4.1.4 Mehr über Tabellennamen

Per Konvention erwartet Active Record als Tabellennamen die Pluralform des zugehörigen Modellnamens. Active Record berücksichtigt dabei einige Sonderformen der englischen Sprache, indem z.B. *person* nach *people* und nicht nach *persons* pluralisiert wird. Tabelle 4.1 gibt einen Überblick über mögliche Kombinationen.

Der von Active Record angenommene Default-Tabellenname kann mit der Methode *set_table_name* überschrieben werden. Die Methode gibt den Tabellennamen einer Modellklasse explizit vor und eignet sich deshalb auch für die Einbindung von Legacy-Datenbanken, in denen die Tabellennamen i.d.R. vorgegeben sind:

```

class Projekt < ActiveRecord::Base
  set_table_name :projekte
end

```

Tabelle 4.1: Abbildung von Modellnamen auf Tabellennamen

Modellklasse	Tabellename
Project	projects
User	users
Person	people
Child	children
Tasks	tasks
Projekt	projekts
Benutzer	benutzers

Einige Datenmodelle verwenden nicht das von Active Record angenommene Pluralisierungsverfahren, sondern nutzen für Tabellennamen genau wie für Modellklassen die jeweilige Singularform des Namens. Trifft dies auch für Ihr Datenmodell zu, kann die automatische Pluralisierung in der Datei `RAILS_ROOT/config/environment.rb` zentral deaktiviert werden:

```
ActiveRecord::Base.pluralize_table_names = false
```

Deutsche Modell- und Tabellennamen

Tabelle 4.1 enthält einige Beispiele für deutsche Klassen- und Tabellennamen. Zugegebenermaßen klingen diese Kombinationen ein wenig merkwürdig, da Active Record deutsche Namen anhand englischer Sprachregeln pluralisiert.

Wenn Sie in Ihrer Anwendung deutsche Modell- und Tabellennamen verwenden wollen, können Sie den ungewünschten Effekt der Rails-Standardpluralisierung durch Anwendung einer der folgenden Strategien vermeiden:

■ Abschalten der Default-Pluralisierung:

```
ActiveRecord::Base.pluralize_table_names = false
```

Die Tabelle der Modellklasse *Projekt* heißt dann per Default *projekt* und nicht *projekts*, was auch für den außenstehenden Leser des Datenmodells einen Sinn ergibt.

■ Zusätzlich bietet Rails die Möglichkeit der expliziten Konfiguration von Singular- und Pluralformen in der Datei `RAILS_ROOT/config/environment.rb`:

```
Inflector.inflections do |inflect|
  inflect.irregular "projekt",      "projekte"
  inflect.irregular "aufgabe",      "aufgaben"
  inflect.irregular "person",       "personen"
  inflect.irregular "unternehmen",  "unternehmen"
end
```

4.2 Active Record direkt verwenden

In Kapitel 3 haben wir Active Record im Kontext einer Rails-Applikation verwendet. Genau so, wie Sie Rails ohne Active Record nutzen, können Sie Active Record auch ohne Rails verwenden. Dies ist sinnvoll, wenn Sie statt Action View ein anderes GUI-Framework (z.B. FXRuby⁴ oder QTRuby⁵ für die Erstellung von Rich-Client-Anwendungen) oder Active Record für Kommandozeilen oder Batch-Programme verwenden wollen.

Kommt Active Record im Kontext einer Rails-Applikation zum Einsatz, übernimmt Rails den Aufbau und die Verwaltung der Datenbankverbindung. Bei einer Verwendung unabhängig von Rails muss Ihr Programm die Verbindung zur Datenbank explizit aufbauen. Hierfür definiert Active Record die Klassenmethode *Base.establish_connection*. Die Methode erwartet eine Hash mit Verbindungsparametern. Das folgende Beispiel zeigt den vollständigen Code, der für die Erstellung eines Standalone Active Record-Programms erforderlich ist:

```
require "rubygems"
require "active_record"

ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host    => "localhost",
  :username => "rails",
  :password => "",
  :database => "activerecord_examples"
)

class Project < ActiveRecord::Base
end

project = Project.new(:name => "Active Record")
project.save
```

4.3 Die Rails-Konsole

Neben der direkten Verwendung von Active Record bietet die Rails-Konsole eine weitere Möglichkeit, die Beispiele dieses Kapitels direkt auszuprobieren. Wechseln Sie dazu in das Verzeichnis eines bestehenden Rails-Projekts (z.B. ontrack), und starten Sie die Konsole:

```
$ cd ontrack
$ ruby script/console
Loading development environment (Rails 2.0.2)
>>
```

⁴ <http://www.fxruby.org/>

⁵ <http://developer.kde.org/language-bindings/ruby/index.html>

Die Konsole lädt die komplette Rails Development-Umgebung und ermöglicht das interaktive Ausprobieren von Ruby-Code im Kontext des aktuellen Projekts, wie z.B. den Umgang mit Active Record-Klassen:

```
Loading development environment (Rails 2.0.2)
>> Project.create(:name => "Wunderloop")
=> #<Project id: 1, name: "Wunderloop",
    created_at: "2008-02-02 11:54:01",
    updated_at: "2008-02-02 11:54:01">
>> Project.create(:name => "Webcert")
=> #<Project id: 2, name: "Webcert",
    created_at: "2008-02-02 11:54:30",
    updated_at: "2008-02-02 11:54:30">
>> Project.find(:all).map(&:name)
=> ["Wunderloop", "Webcert"]
```

Die Konsole kann auch im Zusammenspiel mit Controllern und Actions genutzt werden (vgl. 5.16).

4.4 Objekte erzeugen, laden, aktualisieren und löschen

Die Kernfunktionalität einer jeden Active Record-Klasse wird durch das Akronym *CRUD* beschrieben. *CRUD* steht für *create*, *read*, *update* und *delete* und beschreibt die Erzeugung, das Laden, Aktualisieren und Löschen von Active Record-Objekten. Jede Active Record-Klasse erbt diese Operationen von ihrer Basisklasse *ActiveRecord::Base*.

Um die CRUD-Operationen zu demonstrieren, erzeugen wir zunächst die Active Record-Klasse *Project*:

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

Die zugehörige Tabelle *projects* wird mit folgendem Migrationsskript erzeugt:

```
class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      t.string :name
    end
  end

  def self.down
    drop_table :projects
  end
end
```

4.4.1 Erzeugung

Active Record-Objekte werden durch Aufruf ihres Default-Konstruktors *new* erzeugt:

```
project = Project.new
project.name = "Rails Buch schreiben"
```

Darüber hinaus verfügt jede Active Record-Klasse über einen Konstruktor, der als Parameter eine Hash mit Attributwerten akzeptiert:

```
project = Project.new(:name => "Rails Buch schreiben")
```

Die Verwendung des Hash-basierten Konstruktors eignet sich insbesondere für die direkte Erzeugung von Modellobjekten auf der Basis von Request-Parametern, z.B. wie folgt:

```
project = Project.new(params[:project])
```

Das neue *project*-Objekt existiert bisher nur im Speicher und wird erst nach Aufruf der Methode *save* in die Datenbank geschrieben:

```
project = Project.new(:name => "Rails Buch schreiben")
project.save
```

Der Aufruf erzeugt eine neue Zeile in der Tabelle *projects* und setzt außerdem das Attribut *id* des Objekts *project* auf den von der Datenbank gelieferten Schlüssel:

```
puts project.inspect

=> <Project:0x682384 @attributes={
  "name"=>"Rails Buch schreiben", "id"=>1}>
```

Für die Erzeugung und gleichzeitige Speicherung neuer Objekte bietet Active Record die Methode *create*, die eine Hash mit Attributwerten erwartet:

```
project = Project.create(:name => "Rails Buch schreiben")
assert_not_nil project.id
```

Die Methode *create* eignet sich genau wie die Hash-basierte Version von *new* für die einfache Erzeugung neuer Objekte auf der Basis von Request-Parametern:

```
project = Project.create(params[:project])
```

4.4.2 Objekte laden

Für das Laden von Active Record-Instanzen besitzt jede Active Record-Klasse die statische Methode *find*. Die Methode ist vielfältig parametrisierbar. Der einfachste Anwendungsfall ist das Laden von Objekten über ihre ID:

```
project = Project.find(1)
```

Wird das Objekt mit der übergebenen ID nicht in der Datenbank gefunden, wirft Rails eine *ActiveRecord::RecordNotFound*-Exception.

Die ID-basierte Suche funktioniert sowohl mit einzelnen IDs als auch mit Listen oder Arrays von IDs:

```
projects_by_list = Project.find(1, 2)
projects_by_array = Project.find([1, 2])
```

Das Ergebnis ist in beiden Fällen ein Array mit Modellobjekten bzw. eine *ActiveRecord::RecordNotFound*-Exception in dem Fall, dass mindestens eine der angegebenen IDs nicht existiert.

Weitere mögliche Parameter von *find* sind die Symbole *:first* und *:all*. Der Aufruf

```
Project.find(:first)
```

liefert das erste *Project*-Objekt, das von dem SQL-Statement *select * from projects* zurückgeliefert wird. Hingegen liefert der Aufruf

```
Project.find(:all)
```

ein Array aller *Project*-Objekte, die das SQL-Statement *select * from projects* findet.

Die Optionen und Möglichkeiten der *find*-Methode sind sehr vielfältig und werden deshalb in Abschnitt 4.5 ausführlich beschrieben.

Reload von Objekten

Die Methode *reload* lädt die Attributwerte eines Objekts frisch aus der Datenbank. Wir verwenden die Methode häufig in Unit Tests, um sicherzustellen, dass Modellattribute oder -assoziationen korrekt gespeichert werden:

```
project.iterations << Iteration.new
project.reload
assert_equal 1, project.iterations.length
```

4.4.3 Objekte aktualisieren

Bereits in der Datenbank gespeicherte Objekte werden durch den Aufruf der *save*-Methode aktualisiert:

```
project = Project.new
project.save
```

Die Methode existiert in zwei Varianten: Während *save* einen Booleschen Wert liefert, der anzeigt, ob das Speichern geklappt hat, wirft *save!* bei fehlschlagendem Speicherversuch eine Exception. Der häufigste Grund für das Misslingen eines Speicherversuchs ist eine fehlgeschlagene Validierung. Angenommen, das Attribut *name* darf nicht leer sein (*validates_presence_of :name*, siehe Abschnitt 4.14.1), dann liefert der *save*-Aufruf im folgenden Codeausschnitt den Wert *false*:

```
project = Project.new
puts project.save

$ false
```

Hingegen liefert die Methode *save!* eine *ActiveRecord::RecordInvalid*-Exception:

```
project = Project.new
project.save!

$ ActiveRecord::RecordInvalid: Validation failed: \
  Name can't be blank
```

update_attribute und *update_attributes*

Mit den Methoden *update_attribute* und *update_attributes* bietet Active Record zwei Methoden zum Aktualisieren und Speichern von Modellinstanzen in einem Schritt. Die Methode *update_attribute* erwartet als Parameter den Namen und den Wert des zu aktualisierenden Attributs. Hingegen erwartet *update_attributes* eine Hash mit den zu aktualisierenden Attributwerten.

```
project.update_attribute("name", "Rails Buch schreiben")
project = Project.find(project.id)
assert_equal "Rails Buch schreiben", project.name

project.update_attributes(:name => "Weblog",
                        :description => "...")
```

Die Methode *update_attributes* eignet sich somit besonders zum Aktualisieren von Modellinstanzen auf Basis der Request-Parameter eines vorausgehenden HTTP-Requests:

```
project.update_attributes(params[:project])
```

update

Für das gleichzeitige Laden und Aktualisieren von Objekten stellt Active Record die Klassenmethode *update* zur Verfügung. Die Methode erwartet die ID des zu aktualisierenden Objekts und eine Hash mit Attributwerten:

```
project = Project.update(1, :name => "Weblog")
```

Im Gegensatz zu *save* liefert *update* keinen Booleschen Wert, sondern das aktualisierte Objekt zurück.

update_all

Für die Aktualisierung mehrerer Objekte in einem Schritt besitzt Active Record die Klassenmethode *update_all*. Die Methode erwartet die zu aktualisierenden Attribute mit deren neuen Werten und liefert die Anzahl der aktualisierten Objekte:

```
Task.update_all("priority = 1")
```

Die Menge der zu aktualisierenden Objekte kann dabei über die Angabe optionaler Bedingungen eingeschränkt werden:

```
Task.update_all("priority = 1", "assignee = 'Ralf'")
```

Der zweite Parameter wird hier im where-Teil des zugrunde liegenden SQL-Statements abgesetzt:

```
update tasks set priority = 1 where assignee = 'Ralf'
```

4.4.4 Objekte löschen

Zum Löschen von Modellinstanzen stellt Active Record eine Reihe von Löschmethoden mit unterschiedlichem Verhalten zur Verfügung.

delete

Die Methode *delete* erwartet eine einzelne oder eine Liste von IDs und löscht die entsprechenden Sätze aus der Datenbank:

```
Project.delete(1)
Project.delete([1, 2])
```

delete_all

Die Methode *delete_all* erwartet anstelle von IDs eine Bedingung, welche die zu löschenden Datensätze spezifiziert:

```
Task.delete_all("priority = 4 and assignee = 'Ralf'")
```

Die Bedingung entspricht dabei dem where-Teil eines äquivalenten SQL-Statements.

destroy und *destroy_all*

Die Methoden *destroy* und *destroy_all* funktionieren analog zu den Delete-Methoden. Allerdings kann *destroy* im Gegensatz zu *delete* auch als parameterlose Instanzmethode aufgerufen werden:

```
project = Project.find(1)
project.destroy
```

Bei der Verwendung von *destroy* werden neben der eigentlichen Löschoperation zusätzlich die Active Record-Callbacks ausgeführt. Callbacks sind zwischengeschaltete Methoden, die vor oder nach der eigentlichen Operation aufgerufen werden. Beispielsweise wird der Callback *before_destroy* vor dem Löschen eines Objekts aufgerufen und kann dafür verwendet werden, eventuell noch vorhandene Abhängigkeiten zu prüfen und die Löschoperation ggf. abubrechen. Mehr zum Thema Callbacks finden Sie in Abschnitt 4.15.

4.5 Mehr über Finder

In Abschnitt 4.4.2 haben Sie den grundsätzlichen Mechanismus zum Laden von Active Record-Objekten über deren Primärschlüssel kennengelernt. Da die ID der gewünschten Objekte nicht immer bekannt ist, akzeptiert die Klassenmethode *find* eine ganze Reihe weiterer Optionen zum Aufspüren von Objekten.

4.5.1 Suchbedingungen: *conditions*

Neben den bereits bekannten Parametern *:first* und *:all* dient *:conditions* zur Angabe einer Suchbedingung. Der Parameter akzeptiert ein Array, eine Hash oder einen String mit Suchbedingungen. Die einfachste Form der Suche mit Hilfe des *:conditions*-Parameters ist die Angabe der Suchbedingung in Form des where-Teils einer SQL-Abfrage (ohne das Schlüsselwort *where*):

```
Project.find(:first, :conditions => "name = 'Taskman'")
```

Um mit variablen Werten zu arbeiten, würden Sie diese dann z.B. wie folgt angeben:

```
Project.find(:first, :conditions => "name = '#{name}'")
```

Wir raten Ihnen aber dringend von dieser Schreibweise ab, da sie damit so genanntes SQL-Injection ermöglichen (vgl. 4.5.2). Wir empfehlen Ihnen, die Bedingungen über ein Array anzugeben. Im Falle des Arrays sieht der *find*-Aufruf wie folgt aus:

```
Project.find(:first, :conditions => ["name = ?", name])
```

Das Fragezeichen stellt einen Platzhalter dar, an den Active Record den Wert entsprechend maskiert einfügt. Das resultierende SQL können Sie in der Log-Datei (z.B. *log/development.log*) sehen:

```
SELECT * FROM 'projects' WHERE ('projects'. 'name' = 'CRM')
```

Einzelne Bedingungen können über *and* oder *or* logisch verknüpft werden:

```
Project.find(:first, :conditions => ["name=? and start_date >=?",
                                     name, date])

Project.find(:all,
             :conditions => ["start_date <=? or start_date >=?",
                             "2006-01-01", "2006-05-31"])
```

Bei Suchanfragen mit sehr vielen Parametern können die Bedingungen schnell unübersichtlich werden. Für diesen Fall bietet Active Record die Verwendung von Symbolen an, was die Zuordnung der Suchparameter zu den jeweiligen Platzhaltern in der Bedingung vereinfacht. Die Werte selbst werden dabei als Hash übergeben:

```
Project.find(:all,
             :conditions => ["start_date >= :start and created_at <= :end",
                             {:start => "2006-01-01", :end => "2006-06-01"}])
```

Wenn Sie auf die Gleichheit von Werten prüfen und *nil*, Arrays oder Bereichen als Parameter angeben, wird nicht das erwartete SQL erzeugt. Der folgende Aufruf liefert SQL mit der Bedingung *name=NULL*, die aber *name IS NULL* lauten muss:

```
Project.find(:first, :conditions => ["name=?", nil])
SELECT * FROM 'projects' WHERE (name = NULL) LIMIT 1
```

In diesen Fällen können Sie durch die direkte Angabe einer Hash die Erzeugung des korrekten SQLs erreichen:

```
Project.find(:first, :conditions => {:name => nil})
SELECT * FROM 'projects'
WHERE ('projects'. 'name' IS NULL) LIMIT 1

Task.find(:first, :conditions => {:priority => [1,3]})
SELECT * FROM 'tasks'
WHERE ('tasks'. 'priority' IN (1,3)) LIMIT 1
```


Active Record schließt die zu ersetzenden Strings in Anführungsstriche ein und escaped darüber hinaus alle Zeichen, die eine spezielle Bedeutung für die Datenbank haben.

Alternativ können Sie auch die dynamischen Finder (vgl. 4.6) verwenden, ohne sich Sorgen über etwaige Sicherheitsprobleme machen zu müssen:

```
Person.find_by_name_and_pw(user, pw)
```

4.5.3 Ordnung schaffen: order

Durch Angabe des optionalen Parameters `:order` können Suchergebnisse geordnet werden. Der folgende *find*-Aufruf liefert alle Benutzer mit dem Vornamen „Thomas“, sortiert nach Nachnamen:

```
User.find(:all,
          :conditions => ["firstname = ?", "Thomas"],
          :order => "lastname")
```

Sie können dem Parameter `:order` auch mehrere Spalten übergeben. Wollen Sie die Personenliste zusätzlich nach Geburtsdatum sortieren, dann sieht die zugehörige *find*-Anweisung wie folgt aus:

```
User.find(:all,
          :conditions => ["firstname = ?", "Thomas"],
          :order => "lastname, date_of_birth")
```

Darüber hinaus kann die Sortierrichtung über die Schlüsselwörter *asc* und *desc* vorgegeben werden:

```
User.find(:all,
          :conditions => ["firstname = ?", "Thomas"],
          :order => "lastname asc, date_of_birth desc")
```

4.5.4 Limitieren: limit

Die Anzahl der zurückgelieferten Suchergebnisse lässt sich mit Hilfe des Parameters *:limit* begrenzen:

```
User.find(:all,
          :conditions => ["firstname = ?", "Thomas"],
          :order => "lastname",
          :limit => 5)
```

Der Aufruf liefert uns wieder eine nach Nachnamen sortierte Liste von Personen, die Thomas heißen, diesmal allerdings begrenzt auf maximal 5 Einträge.

4.5.5 Seitenweise: limit und offset

Der Parameter *:limit* kann in Kombination mit *:offset* für das seitenweise Anzeigen von Suchergebnissen verwendet werden. Angenommen, Sie wollen sämtliche

Benutzer Ihres Systems seitenweise zu je 10 Benutzern pro Seite anzeigen. Der zugehörige *find*-Aufruf sieht dann folgendermaßen aus:

```
User.find(:all,
  :order => "lastname",
  :limit => 10, :offset => 10 * pages)
```

Der Aufrufer der oben stehenden *find*-Methode muss sich dabei selbst um die Verwaltung der Variablen *pages* kümmern.

4.5.6 Weitere Parameter: joins und include

Aus Ihrer bisherigen Erfahrung mit SQL-Datenbanken wissen Sie sicherlich, dass „richtige“ Datenabfragen häufig mehr als eine Tabelle mit einbeziehen, und fragen sich, ob das auch mit Active Record möglich ist. Insbesondere die Tatsache, dass Finder-Methoden immer Objekte desselben Typs zurückliefern, mag hier anfänglich verwirrend erscheinen.

:joins

Als Beispiel für den *:joins*-Parameter soll eine Funktion dienen, die eine Liste aller Projektleiter von Ruby-Projekten ausgibt.⁶ Projekte sind in der Tabelle *projects* und Projektleiter in der Tabelle *people* gespeichert. Bei direkter Verwendung von SQL würde der folgende SQL-Befehl das gewünschte Ergebnis liefern:

```
select * from projects, people where projects.manager_id =
  people.id and projects.name like '%Ruby%'
```

Ein äquivalenter Active Record-Aufruf sieht wie folgt aus, wobei die verwendete join-Syntax datenbankspezifisch (hier MySQL) ist:

```
ruby_leads = Person.find(
  :all,
  :conditions => "pr.name like '%Ruby%'",
  :joins => "as pe inner join projects as pr on " +
    "pe.id = pr.manager_id")
```

Das zurückgelieferte Array enthält Objekte vom Typ *Person*, die zusätzlich um projektspezifische Attribute, wie *name* oder *begin*, erweitert wurden:

```
puts ruby_leads[0].inspect

$ #<Person:0x6eb140 @attributes={"name"=>"Ruby on Rails",
  "manager_id"=>"1", "begin"=>nil, "salary"=>nil,
  "type"=>nil, "id"=>"1", "firstname"=>"Ralf",
  "end"=>nil, "description"=>nil, "company_id"=>nil,
  "budget"=>nil, "parent_id"=>nil}>
```

Zugegeben, das Beispiel wirkt ein wenig konstruiert. Das liegt daran, dass Active Record fast alle benötigten Relationen auf Objekt-Ebene handhabt, sodass der Parameter *:joins* nur sehr selten wirklich benötigt wird.

⁶ Der Einfachheit halber nehmen wir an, dass ein Ruby-Projekt dadurch gekennzeichnet ist, dass der Projektname das Wort „Ruby“ enthält.

:include

Der Parameter *:include* gibt die Assoziationen an, die in einem *find*-Aufruf initial mitgeladen werden sollen. Im folgenden Beispiel werden die Iterationen eines Projekts erst beim ersten Zugriff geladen, was dem Default-Verhalten von Active Record entspricht (lazy loading):

```
project = Project.create(:name => "Rails")
project.iterations.create(:name => "I1")
projects = Project.find(:all) # Iteration nicht geladen
```

Die Verwendung des Parameters *:include* bewirkt ein „Vorabladen“ der angegebenen Assoziation:

```
projects = Project.find(
  :all,
  :include => :iterations) # Iteration werden geladen
```

Während im ersten Beispiel zwei SQL-Abfragen benötigt werden, lädt Active Record sämtliche Objekte des zweiten Beispiels in einer einzigen SQL-Abfrage.

4.6 Dynamische Finder

Für Entwickler mit einer SQL-Aversion bietet Active Record Attribut-basierte dynamische Finder. Ihr Name ist hier tatsächlich deutlich komplizierter als ihre Benutzung. Dazu ein Beispiel. Statt

```
User.find(:first,
  :conditions => ["lastname = ?", "Fowler"])
```

können wir die folgende Anweisung schreiben:

```
User.find_by_lastname("Fowler")
```

Das Ergebnis ist das gleiche, mit dem Unterschied, dass der zweite Aufruf deutlich einfacher zu schreiben (ich schreibe es hin, wie ich es denke) und darüber hinaus auch deutlich einfacher zu lesen ist. Die Intention des Codes wird auf den ersten Blick klar. Ein zusätzlicher Kommentar ist nicht erforderlich.

Dynamische Finder starten immer mit *find.by* oder *find.all.by*, gefolgt von einem oder mehreren Attributen. Die *all*-Variante liefert ein Array, während der Aufruf ohne *all* ein einzelnes Objekt liefert. Die Attribute dynamischer Finder können mit Hilfe logischer Operatoren beliebig verknüpft werden:

```
User.find_by_lastname_and_profession("Fowler", "Writer")
```

Dynamische Finder werden von Active Record bei ihrer erstmaligen Benutzung erzeugt. Wenn die Methode aufgerufen wird, konvertiert Active Record den Aufruf in einen *find*-Aufruf mit entsprechenden Parametern.

Schreiben Sie Ihre Suchabfrage einfach hin, wie sie Ihnen in den Kopf kommt. Die Wahrscheinlichkeit, dass sie funktioniert, ist ziemlich groß. Kann die Methode nicht generiert werden, teilt Ihnen Active Record dies mit einer *NoMethodError*-Exception mit.

4.7 Kann ich weiterhin SQL verwenden?

Sie können. Active Record kapselt zwar viele SQL-Details in der *find*-Methode, erlaubt aber weiterhin die direkte Verwendung von SQL. Die Verwendung von *find* ist für viele Datenbankabfragen der einfachere und zu bevorzugende Weg, doch ist gerade im Hinblick auf komplexe oder Performance-kritische SQL-Anweisungen die direkte Verwendung von SQL eine wichtige und notwendige Alternative.

find_by_sql

Für die direkte Verwendung von SQL bietet Active Record die statische Methode *find_by_sql*, die einen SQL-Befehl direkt an die Datenbank absetzt:

```
Project.find_by_sql("select * from projects")
```

Der Aufruf liefert ein Array mit allen gespeicherten Projekten. Was aber passiert, wenn die Tabelle des Select-Statements nicht mit der Active Record-Klasse korrespondiert, auf der *find_by_sql* aufgerufen wird? Das folgende Beispiel beantwortet diese Frage:

```
result = Project.find_by_sql("select * from iterations")
puts result[0].inspect
```

```
$ <Project:0x65c8c8 @attributes={"name"=>"I1",
  "project_id"=>nil, "id"=>"1"}>
```

Active Record liefert eine Liste von Objekten, deren Attribute und Methoden mit den in der Ergebnismenge des Select-Aufrufs enthaltenen Feldern übereinstimmen. Die zurückgelieferten Objekte sind Instanzen der Klasse, auf der *find_by_sql* aufgerufen wird, in diesem Fall also *Project*-Instanzen. Allerdings erweitert Active Record die Objekte um Attribute und Methoden zum Zugriff auf die in der SQL-Ergebnismenge enthaltenen Spalten.

Das Prinzip lässt sich gut am Beispiel von Abfragen über mehrere Tabellen verdeutlichen. Das folgende Beispiel liefert uns eine Liste von Projekten mit den Daten ihrer jeweiligen Manager:

```
result = Project.find_by_sql(
  "select * from projects, people " +
  "where projects.manager_id = people.id")
puts "Project: " + result[0].name
puts "Manager: " + result[0].firstname
```

```
$ Project: Rails Buch schreiben
  Manager: Ralf
```

Was passiert aber, wenn gejointe Tabellen die gleichen Felder aufweisen? Z.B. haben die Tabellen *projects* und *iterations* beide das Feld *name*:

```
result = Project.find_by_sql("select * from projects, " +
  "iterations where iterations.id = iterations.project_id")
puts result[0].inspect
$ <Project:0x66c304 @attributes={"name"=>"Iteration 1",
```

```
"project_id"=>"1", "manager_id"=>"1", "begin"=>nil,
"id"=>"1", "end"=>nil}>
```

Die Objekte des Ergebnisarrays besitzen nur ein Attribut *name*, dessen Wert dem Feld *name* der zuletzt im SQL-Statement aufgeführten Tabelle entspricht (*iterations*).

Anstelle eines Strings können Sie *find_by_sql* auch ein Array übergeben:

```
Project.find_by_sql(["select * from projects where " +
                    "name like ?", "%ails"])
```

4.8 Metadaten – Daten über Daten

Manchmal ist es wichtig, neben den Inhalten einer Tabelle zusätzlich Informationen über deren Aufbau und Struktur zu erfahren. Active Record bietet dafür eine Reihe statischer Methoden zum Zugriff auf die Metadaten einer Active Record-Klasse.

columns

Die Methode *columns* liefert ein Array mit den Spaltenobjekten der zugehörigen Tabelle:

```
for col in User.columns
  puts col.inspect
end

$ #<ActiveRecord::ConnectionAdapters::Column:0x6567ac
  @limit=100, @default=nil, @type=:string, @name="name">
```

column_names

Die Methode *column_names* liefert ein Array mit den Spaltennamen der zugehörigen Tabelle:

```
for col_name in User.column_names
  puts col_name
end
```

content_columns

Während *column_names* sämtliche Spalten einer Tabelle liefert, erbringt die Methode *content_columns* nur die Inhalts-bezogenen Spaltenobjekte einer Tabelle. Dies ist eine Liste, aus der die Spalte *id* sowie alle Spalten, die mit *_id* oder *_count* enden, entfernt wurden. Statisches Scaffolding nutzt die Methode beispielsweise für die dynamische Ausgabe von Modellattributen:

```
<% for column in Project.content_columns %>
<p>
  <b><%= column.human_name %>:</b>
  <%=h @project.send(column.name) %>
</p>
<% end %>
```

4.9 Assoziationen

Active Record-Assoziationen ermöglichen die Modellierung von Beziehungen zwischen Modellklassen. Die Programmierung der Beziehungen vollzieht sich immer in zwei Schritten: Zum einen muss die Beziehung in der Datenbank mit Hilfe von Fremdschlüsseln definiert, und zum anderen müssen die Beziehungen auf Modellebene mit Hilfe spezieller Active Record-Deklarationen beschrieben werden. Active Record unterscheidet drei Assoziationstypen:

- 1:1-Beziehungen
- 1:N-Beziehungen
- N:M-Beziehungen

4.9.1 Grundsätzliches

Bevor wir in die Details der einzelnen Assoziationstypen einsteigen, wollen wir Ihnen die grundsätzliche Funktionsweise von Rails-Assoziationen am Beispiel einer 1:N-Assoziation erläutern: Parent –* Children.

```
class CreateParentsAndChildren < ActiveRecord::Migration
  def self.up
    create_table :parents do |t|
      t.string :name
    end

    create_table :children do |t|
      t.string :name
      t.integer :parent_id
    end
  end

  def self.down
    drop_table :parents
    drop_table :children
  end
end
```

Die Tabellendefinition modelliert die Beziehung vollständig: Die Tabelle *children* definiert den Fremdschlüssel *parent_id* und stellt so die 1:N-Verbindung zwischen Parent und Children her. Allerdings können Sie die Assoziationen noch nicht wirklich bequem benutzen, es sei denn, Sie sind SQL-Experte und wollen neue Objekte und deren Beziehung manuell mit dem MySQL-Client bearbeiten. Was wir wollen, ist die Benutzung der Modellassoziationen auf Modellebene, und hier kommen die Active Record-Assoziationsmakros ins Spiel.

Assoziationsmakros sind statische Methoden, die eine Modellklasse zur Laufzeit dynamisch um eine Reihe zusätzlicher Methoden erweitern. Wenn Sie z.B. in der Klasse *Parent* auf die assoziierten *Child*-Objekte zugreifen wollen, dann benötigt *Parent* einen Aufruf der *has_many*-Methode:

```
class Parent < ActiveRecord::Base
  has_many :children
end
```

Active Record erweitert *Parent* um Methoden zum Zugriff auf die assoziierten Child-Objekte. Eine dieser Methoden ist z.B. *children*, die ein Array der zugehörigen Children liefert:

```
parent = Parent.find(1)
parent.children.map(&:name)
```

Für den Zugriff eines Child-Objekts auf sein Parent-Objekt stellt Active Record die *belongs_to*-Methode zur Verfügung:

```
class Child < ActiveRecord::Base
  belongs_to :parent
end
```

Auch hier erweitert Active Record die Child-Klasse um Methoden zur Verwendung des assoziierten Parents:

```
parent = Parent.find(1)
parent.children.each do |child|
  puts child.parent.name
end
```

Wie gesagt, der Aufruf der Assoziationsmakros ist keine Pflicht. Sie müssen von Fall zu Fall entscheiden, welches Modell einer Assoziation Zugriff auf das jeweils andere Modell benötigt. Beispielsweise kann es gut sein, dass ein Parent auf seine Child-Objekte zugreifen muss, es aber keinen Grund für den Zugriff eines Child-Objekts auf seinen Parent gibt. In diesem Fall kann die *belongs_to*-Deklaration in der Klasse *Child* entfallen.

Warum werden Beziehungen nicht aus der Datenbankdefinition abgeleitet?

Assoziationen werden an zwei Stellen definiert: in der Datenbank und in den Modellklassen. In Vorträgen und Schulungen werden wir bezüglich dieses Punktes immer wieder gefragt, ob es sich hier nicht um die Verletzung des DRY-Prinzips handelt. Auf den ersten Blick mag das auch so aussehen: Aus dem Fremdschlüssel der Datenbanktabelle müssten sich doch eigentlich die entsprechenden Assoziationsattribute und -methoden auf Modellebene erzeugen lassen. Auf den zweiten Blick stellt man dann aber fest, dass sich 1:1- und 1:N-Beziehungen in der Tabellendefinition nicht unterscheiden: Beide zeichnen sich durch einen Fremdschlüssel in der assoziierten Tabelle aus. Demzufolge fehlt Rails hier die notwendige Information, welcher Assoziationstyp denn nun auf Modellebene gewünscht ist. Die zusätzliche Deklaration ist also erforderlich und keine Verletzung des DRY-Prinzips.

4.9.2 1:1-Beziehungen: `has_one` – `belongs_to`

Als Beispiel für eine 1:1-Relation dient uns ein Projekt mit seinem Projektplan (Schedule): Ein Projekt hat genau einen Projektplan, und ein Projektplan bezieht sich auf genau ein Projekt (siehe Graphik 4.1).

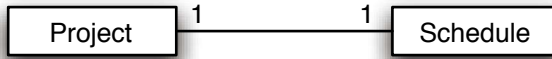


Abbildung 4.1: Ein Projekt hat genau einen Projektplan

Im Folgenden werden wir die Klasse *Project* als Parent-Klasse und die Klasse *Schedule* als Child-Klasse der Relation bezeichnen. Die Child-Klasse – oder besser: die Datenbanktabelle der Child-Klasse – hält den Fremdschlüssel der Relation. Das Migrationsskript zum Erzeugen der Tabelle *Schedules* sieht wie folgt aus:

```

class CreateSchedules < ActiveRecord::Migration
  def self.up
    create_table :schedules do |t|
      t.string :name
      t.date :begin, :end
      t.integer :project_id
    end
  end

  def self.down
    drop_table :schedules
  end
end

```

Auf Modellebene wird die 1:1-Relation durch die beiden Assoziationsmakros *has_one* und *belongs_to* modelliert.

Die „*has_one*“-Deklaration

Die Deklaration *has_one* modelliert die Sicht der Parent-Klasse, d.h. der Klasse, die das aggregierte Child-Objekt enthält. In unserem Beispiel modelliert *has_one* also die Sicht der Klasse *Project*. Die Deklaration benötigt als einzigen Pflichtparameter den Namen der Assoziation:

```

class Project < ActiveRecord::Base
  has_one :schedule
end

```

Die *has_one*-Deklaration veranlasst Active Record zur dynamischen Erweiterung der Klasse *Project* um eine Reihe zusätzlicher Methoden, die die Benutzung der Assoziation auf Modellebene ermöglichen. Die Methoden werden im Folgenden am konkreten Beispiel des Schedules beschrieben.

schedule=

Die Methode *schedule=* weist dem Projekt einen neuen Plan zu:

```
project = Project.new(:name => "Taskman")
project.schedule = Schedule.new(:name => "Taskschedule")
project.save
```

Der Aufruf *project.save* speichert nicht nur das *Project*-Objekt, sondern auch die erzeugte *Schedule*-Instanz.

schedule

Die Methode liefert den Projektplan eines Projekts. Der Projektplan wird übrigens erst aus der Datenbank geladen, sobald *schedule* das erste Mal aufgerufen wird (lazy loading):

```
reloaded = Project.find(project.id)
puts reloaded.schedule.name

$ Taskschedule
```

create_schedule

Die Methode erzeugt eine neue *Schedule*-Instanz und weist sie dem Projekt zu. Die neu erzeugte *Schedule*-Instanz wird sofort gespeichert, sodass ein expliziter *save*-Aufruf entfallen kann.

```
project = Project.create(:name => "Taskman")
project.create_schedule(:name => "Taskschedule")
```

build_schedule

Die Methode erzeugt analog zu *create_schedule* eine neue *Schedule*-Instanz und weist sie dem Projekt zu. Während *create_schedule* das neu erzeugte *Schedule*-Objekt sofort speichert, muss bei Verwendung von *build_schedule* ein expliziter *save*-Aufruf erfolgen:

```
project = Project.create(:name => "Taskman")
project.build_schedule(:name => "Taskschedule")
project.save
```

Weitere Parameter der „has_one“-Deklaration

Die *has_one*-Methode benötigt als einzigen Pflichtparameter den Namen der Assoziation. Darüber hinaus akzeptiert die Methode eine Reihe optionaler Parameter.

:class_name

Mit dem Parameter *:class_name* wird der Name der assoziierten Klasse angegeben. Dies ist erforderlich, wenn der Klassenname nicht mit dem Namen der Assoziation übereinstimmt. Wollen Sie z.B. die Beziehung zwischen einer Firma (Company) und ihrem Geschäftsführer (Chief) modellieren, dann sollte der Name der Assoziation auch deren Intention ausdrücken und *:chief* heißen. Da Chefs, genau wie andere Mitarbeiter der Firma, in der Tabelle *people* gespeichert werden, ist für diese Assoziation die explizite Angabe des Klassennamens *Person* erforderlich:


```
class Company < ActiveRecord::Base
  has_one :chief, :class_name => "Person"
end
```

Der zusätzliche Parameter *class_name* sorgt dafür, dass die neuen Chief-Methoden Instanzen der Klasse *Person* akzeptieren und zurückliefern:

```
company = Company.new
company.chief = Person.new(:firstname => "Steve")
```

:conditions

Die Company-Chief-Beziehung ist auch ein gutes Beispiel für den *:conditions*-Parameter. Der Parameter dient der Angabe von bestimmten Bedingungen, die das assoziierte Objekt erfüllen muss. Dazu ein Beispiel: Damit eine Person überhaupt für den Job eines Geschäftsführers in Frage kommt, muss sie mindestens 5 Jahre im Unternehmen arbeiten. Diese Bedingung können wir in Active Record folgendermaßen formulieren:

```
class Company < ActiveRecord::Base
  has_one :chief,
    :class_name => "Person",
    :conditions => "years_of_employment >= 5"
end
```

Der *:conditions*-Parameter des obigen Codeausschnitts bewirkt, dass ein assoziierter Firmenchef nur dann geladen wird, wenn er seit mindestens 5 Jahren in dem Unternehmen arbeitet. Ein Chef, der erst 4 Jahre im Unternehmen arbeitet, bleibt beim Laden seiner Firma unberücksichtigt:

```
company = Company.new
company.build_chief(:firstname => "Peter",
                  :years_of_employment => 4)

company.save
reloaded = Company.find(company.id)
assert_nil reloaded.chief
```

:order

Der *:order*-Parameter mag auf den ersten Blick vielleicht ein wenig merkwürdig erscheinen (Ordnung bei nur einem Objekt?), wird aber interessant, wenn die 1:1-Beziehung keine Entsprechung in der darunterliegenden Datenbank hat.

Angenommen, für ein Projekt werden monatlich Statusreports geschrieben. Diese Beziehung zwischen Projekt und Reports könnten wir mit Hilfe einer 1:N-Beziehung modellieren. Darüber hinaus existiert eine 1:1-Beziehung zwischen dem Projekt und dem jeweils aktuellen Report. Diese Beziehung ist zwar nicht in der darunterliegenden Datenbank enthalten, lässt sich aber trotzdem auf Modellebene folgendermaßen abbilden:

```
class Project < ActiveRecord::Base
  has_one :current_report, :class_name => "Report",
    :order => "created desc"
end
```

Der jeweils aktuelle Statusreport kann nun über den Aufruf der Getter-Methode *current_report* abgefragt werden:

```
project = Project.find(1)
puts project.current_report.name
```

:dependent

Der Parameter *:dependent => :destroy* bewirkt, dass das Child-Objekt gelöscht wird, wenn entweder das Parent-Objekt gelöscht oder dem Parent-Objekt ein neues Child-Objekt zugewiesen wird. Beispielsweise kann ein Projektplan nicht ohne ein zugehöriges Projekt existieren:

```
class Project < ActiveRecord::Base
  has_one :schedule, :dependent => :destroy
end
```

Die Wirkung des *:dependent*-Parameters können Sie durch folgendes Codefragment überprüfen:

```
project = Project.create(:name => "Taskman")
project.create_schedule(:name => "Taskschedule")
schedule_id = project.schedule.id
project.destroy

Schedule.find(schedule_id)

$ ActiveRecord::RecordNotFound: Couldn't find Schedule
  with ID=1
```

Nach dem Löschen des Projekts wird der zugehörige Projektplan ebenfalls nicht mehr in der Datenbank gefunden, d.h. Active Record hat den Plan automatisch gelöscht.

:foreign_key

Ausgehend vom Namen der Parent-Klasse, die die *has_one*-Deklaration enthält, schließt Active Record auf den Namen des Fremdschlüssels in der Tabelle des assoziierten Child-Objekts. Dazu erweitert Active Record den Assoziationsnamen um das Suffix *.id*. Aus dem Klassennamen *Project* wird der in der Tabelle *schedules* als Fremdschlüssel enthaltene Name *project_id*.

Besteht die Notwendigkeit, von dieser Konvention abzuweichen, können Sie den Fremdschlüssel explizit durch Angabe des Parameters *:foreign_key* vorgeben. Heißt der Fremdschlüssel in der Tabelle *schedules* nicht *project_id*, sondern *related_project*, dann muss die *has_one*-Deklaration in der Klasse *Project* wie folgt geändert werden:

```
class Project < ActiveRecord::Base
  has_one :schedule, :foreign_key => "related_project"
end
```

Die „belongs_to“-Deklaration

Auf der Gegenseite, d.h. auf Seite der Child-Klasse *Schedule*, wird die 1:1-Relation durch die Deklaration *belongs_to* modelliert. Die Deklaration benötigt als Parameter den Namen der Assoziation:

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

Die Deklaration erweitert die Klasse *Schedule* um eine Reihe von Methoden, die wir im Folgenden beschreiben.

project=

Die Setter-Methode *project=* ermöglicht die Zuweisung eines Projekts zu einem *Schedule*. Das Herstellen der Verbindung in einer 1:1-Relation ist also von beiden Seiten möglich. Auch das Speicherverhalten ist identisch: Egal, welches Objekt der Assoziation gespeichert wird, das assoziierte Objekt wird automatisch mit gespeichert:

```
schedule = Schedule.create(:name => "Taskschedule")
project = Project.new(:name => "Taskman")
schedule.project = project
schedule.save # speichert auch das Projekt
```

project

Die Getter-Methode liefert das assoziierte Projekt. Auch hier gilt: Das assoziierte Projekt wird erst bei der erstmaligen Benutzung von *project* geladen:

```
schedule = Schedule.find(schedule.id)
assert_equal "Taskman", schedule.project.name
```

project.nil?

Prüft, ob ein assoziiertes Objekt vorhanden ist:

```
schedule = Schedule.create(:name => "Taskschedule")
assert schedule.project.nil?
```

create_project

Erzeugt ein neues *Project*-Objekt und assoziiert es mit diesem *Schedule*-Objekt. Das neu erzeugte *Project*-Objekt ist nach Aufruf dieser Methode gespeichert. Die Methode liefert das erzeugte Objekt zurück:

```
schedule = Schedule.create(:name => "Schedule")
schedule.create_project(:name => "On Track")
```

build_project

Erzeugt ein neues *Project*-Objekt und assoziiert das Objekt mit diesem *Schedule*-Objekt. Das *Project*-Objekt ist nach Aufruf dieser Methode nicht gespeichert, sodass ein expliziter *save*-Aufruf auf *Schedule* notwendig ist:

```
schedule = Schedule.create(:name => "Schedule")
schedule.build_project(:name => "On Track")
schedule.save
schedule.reload
assert_not_nil schedule.project
```

Weitere Parameter der „belongs_to“-Deklaration

Die *belongs_to*-Deklaration erwartet als einzigen Pflichtparameter den Namen der Assoziation. Darüber hinaus stehen eine Reihe weiterer Parameter für die Anpassung der Deklaration zur Verfügung, z.B. bei Abweichung vom Standardfall.

:class_name

Name der assoziierten Klasse, sofern dieser vom Namen der Assoziation abweicht.

:conditions

Bedingungen, unter denen das assoziierte Projekt geladen wird.

:order

Reihenfolge assoziierter Objekte. Dieser Parameter ist insbesondere für Assoziationen sinnvoll, die keine Entsprechung in der darunterliegenden Datenbank besitzen.

:foreign_key

Per Konvention heißt der Fremdschlüssel so wie die referenzierte Tabelle, gefolgt von *_id*. In unserem Beispiel halten wir uns an diese Konvention, indem wir den in der Tabelle *schedules* enthaltenen Fremdschlüssel *project_id* genannt haben. Wenn Sie von dieser Konvention abweichen, dann müssen Sie den Fremdschlüssel explizit mit Hilfe des Parameters *:foreign_key* angeben.

:counter_cache => true

Liefert die Anzahl der assoziierten Child-Objekte. Für 1:1-Relationen ist diese Anzahl immer 0 oder 1. Dieser Parameter ist insbesondere für 1:N-Beziehungen sinnvoll und wird deshalb erst in Abschnitt 4.9.3 genauer beschrieben.

Gibt es einen Unterschied zwischen „has_one“ und „belongs_to“ ?

Bei einer 1:1-Beziehung ist es prinzipiell egal, welche der beiden assoziierten Klassen die *has_one*- oder die *belongs_to*-Deklaration enthält. Als generelle Regel gilt: Der Fremdschlüssel befindet sich immer in der Tabelle, deren zugehöriges Objekt die *belongs_to*-Assoziation enthält. D.h. *has_one* geht immer davon aus, dass sich der Fremdschlüssel in der jeweils anderen Tabelle befindet.

Außerdem gilt: Per *belongs_to* verbundene Objekte werden beim Speichern des Parent-Objekts automatisch validiert, während dies für per *has_one* verbundene Objekte nicht gilt. Konkret bedeutet das: Sie dürfen keine zwei Objekte mit gegenseitigen *belongs_to*-Assoziationen verbinden, da Active Record andernfalls beim Speichern in eine Endlosschleife gerät.

1:1-Assoziation: Deklaration und erzeugte Methoden

Tabelle 4.2 fasst die Deklaration von 1:1-Assoziationen sowie die von der Deklaration erzeugten Methoden zusammen.

Tabelle 4.2: 1:1-Assoziation: Deklaration und erzeugte Methoden

1:1-Assoziation	Parent: Project	Child-Klasse: Schedule
Deklaration	has_one()	belongs_to()
erzeugte Methoden	schedule() schedule=() create_schedule() build_schedule()	project() project=() project.nil?() create_project() build_project()

4.9.3 1:N-Beziehungen: has_many – belongs_to

Eine 1:N-Beziehung modelliert eine Assoziation, in der ein Parent-Objekt in Relation zu beliebig vielen Child-Objekten steht. Als Beispiel dient uns ein Projekt, das aus beliebig vielen Iterationen besteht (siehe Abbildung 4.2).

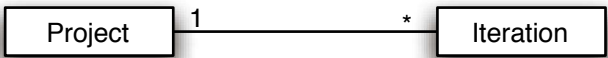


Abbildung 4.2: Ein Projekt hat mehrere Iterationen.

Das folgende Migrationsskript erzeugt die benötigten Datenbanktabellen (*projects* ist nicht mehr enthalten, da weiter oben bereits beschrieben). Die 1:N-Assoziation wird dabei durch den Fremdschlüssel *project_id* in der Tabelle *iterations* modelliert:

```
class CreateIterations < ActiveRecord::Migration
  def self.up
    create_table :iterations do |t|
      t.string :name
      t.date :begin, :end
      t.integer :project_id
    end
  end

  def self.down
    drop_table :iterations
  end
end
```

Auf Modellebene werden 1:N-Relationen durch die beiden Deklarationen *has_many* und *belongs_to* modelliert:

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

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

Beide Deklarationen erwarten als einzigen Pflichtparameter den Namen der Assoziation.

Die von „has_many“ erzeugten Methoden

Analog zu *has_one* erweitert Active Record die *Project*-Klasse um eine Reihe von Methoden für die Benutzung der 1:N-Relation auf Modellebene. Die Methoden werden im Folgenden am konkreten Beispiel der *iterations*-Assoziation beschrieben.

iterations=

Weist dem Projekt ein Array mit Iterationen zu:

```
project = Project.create(:name => "OnTrack")
project.iterations = [Iteration.new(:name => "I1")]
project.save
```

Die Iterationen eines Projekts werden beim Hinzufügen automatisch mit gespeichert. Der Aufruf von *project.save* ist im obigen Beispiel also optional. Anders sieht es hingegen aus, wenn die Project-Instanz an Stelle von *Project.create* mit *Project.new* erzeugt wird: In diesem Fall werden weder das Projekt noch die Iterationen automatisch gespeichert, sofern kein expliziter *project.save*-Aufruf erfolgt.

iterations

Liefert eine Liste mit den Iterationen des Projekts. Die Iterationen werden erst dann geladen, wenn auf der Methode *iterations* selbst das erste Mal Methoden aufgerufen werden.

```
project.iterations          # kein Laden
project.iterations.first.name # Laden
```

iterations<<

Fügt eine neue Iteration dem Ende der Iterationsliste eines Projekts an:

```
project.iterations << Iteration.new(:name => "I2")
```

Ein expliziter Aufruf von *project.save* kann im obigen Beispiel entfallen; die neue Iteration wird unmittelbar nach ihrer Erzeugung und dem Aufruf *;;* in die Datenbank geschrieben:

```
project.iterations << Iteration.new(:name => "I2")
project.reload
assert_equal 2, project.iterations.length
```

iterations.delete

Löscht die Verbindung zwischen einer oder mehreren Iterationen und ihrem Projekt:

```
project = Project.create(:name => "Taskman")
i1 = Iteration.new(:name => "I1")
i2 = Iteration.new(:name => "I2")
project.iterations << i1 << i2
project.iterations.delete(i1, i2)
assert project.iterations.empty?
assert_equal 2, Iteration.find(:all).length
```

Die Iterationen werden nicht aus der Datenbank gelöscht, stattdessen wird ausschließlich deren Fremdschlüssel *project_id* auf *nil* gesetzt.

iterations.empty?

Prüft, ob das Projekt assoziierte Iterationen besitzt.

iterations.clear

Entfernt alle Iterationen aus dem Projekt, löscht sie jedoch nicht aus der Datenbank.

iterations.size

Die Methode *size* liefert die Anzahl der mit dem Projekt assoziierten Iterationen.

iterations.find

Die Methode durchsucht die Iterationsliste eines Projekts nach einer oder mehreren den Suchkriterien entsprechenden Iterationen. Die Parameter entsprechen denen der Methode *ActiveRecord::Base.find* (siehe Abschnitt 4.5).

iterations.build

Die Methode erzeugt ein neues Iterations-Objekt und fügt es dem Ende der Iterationen des Projekts an. Die hinzugefügte Iteration wird nicht gespeichert.

```
project = Project.create(:name => "Taskman")
project.iterations.build("name" => "I1")
assert !project.iterations.empty?
project.reload
assert project.iterations.empty?
```

iterations.create

Erzeugt und speichert eine neue Iteration und schließt die Iteration an das Ende der Iterationsliste des Projekts an.

```
project = Project.create(:name => "OnTrack")
project.iterations.create("name" => "I1")
project.reload
assert !project.iterations.empty?
assert_equal 1, Iteration.find(:all).length
```

Die Parameter der „has_many“-Deklaration

Analog zu *has_one* akzeptiert *has_many* eine ganze Reihe von optionalen Parametern zur Anpassung der Assoziation. Die Parameter *:class_name*, *:conditions*, *:order* und *:foreign_key* entsprechen denen der *has_one*-Deklaration (siehe Abschnitt 4.9.2). Deshalb erklären wir an dieser Stelle nur die noch nicht beschriebenen Parameter *:dependent*, *:finder_sql* und *:counter_sql*.

:dependent

Der Parameter *:dependent* bewirkt das automatische Löschen assoziierter Iterationsobjekte, wenn das zugehörige Projektobjekt gelöscht wird. Bei *:dependent => :destroy* werden die assoziierten Iterationsobjekte zunächst instanziiert und anschließend durch Aufruf ihrer *destroy*-Methode gelöscht. Sofern vorhanden, werden dabei die für Iterationen definierten Callbacks *before_destroy* und *after_destroy* ausgeführt.

Hingegen erfolgt bei *:dependent => :delete_all* das Löschen der Iterationsobjekte direkt auf der Datenbank. Die Objekte werden nicht instanziiert und eventuell vorhandene Callbacks nicht ausgeführt. Die Verwendung von *:delete_all* ist effizienter und dann zu empfehlen, wenn keine Aufräumarbeiten vor oder nach dem Löschen einer Iteration erforderlich sind. Mehr zum Thema Callbacks finden Sie in Abschnitt 4.15.

Ein weiterer gültiger Wert für *:dependent* ist das Symbol *:nullify*. Bei *:dependent => :nullify* werden die Iterationsobjekte nicht aus der Datenbank gelöscht, sondern es wird ausschließlich deren Fremdschlüssel auf *NULL* gesetzt. Die Iterationen sind weiterhin vorhanden, doch nicht mehr mit ihrem ursprünglichen Projekt assoziiert.

:finder_sql

Mit diesem Parameter können Sie das exakte für das Laden der assoziierten Iterationen verwendete SQL-Statement vorgeben. Wir empfehlen die Verwendung dieses Parameters insbesondere bei komplexen Assoziationen, wie z.B. solchen, die über mehrere Tabellen gehen.

:counter_sql

Dieser Parameter gibt das SQL-Statement zum Ermitteln der Anzahl der assoziierten Objekte vor.

Die „belongs_to“-Deklaration

Die *belongs_to*-Deklaration modelliert in einer 1:N-Relation die N-Seite, d.h. die Child-Seite der Relation. Auch hier kann man sich wieder gut merken, dass die Klasse mit der *belongs_to*-Deklaration die Klasse mit dem Fremdschlüssel ist.

Die Sicht eines Child-Objekts einer 1:N-Relation ist absolut identisch mit der Sicht eines Child-Objekts einer 1:1-Relation. D.h., für beide Objekte befindet sich auf der jeweils anderen Seite genau ein einziges Parent-Objekt: Für den Schedule ist das ebenso wie für eine Iteration in unserem Beispiel ein Projekt. Deklaration, Parameter und Verhalten der *belongs_to*-Relation sind identisch, unabhängig davon, ob sie in einer 1:1- oder 1:N-Beziehung genutzt werden:


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

Der Iteration stehen jetzt die bereits in Abschnitt 4.9.2 beschriebenen Methoden zum Zugriff und Bearbeiten des assoziierten Projekts zur Verfügung.

Reihenfolge und Counter-Caching

In Abschnitt 4.9.2 haben wir die Parameter der *belongs_to*-Deklaration beschrieben. Zweien dieser Parameter kommt im Kontext einer 1:N-Beziehung eine besondere Bedeutung zu: *:order* und *:counter_cache*.

:order

Der Parameter bestimmt die Reihenfolge der Child-Objekte. Die Angabe der Reihenfolge erfolgt im „order by“-Format. Beispielsweise sollen die Iterationen unseres Projektmanagement-Systems nach Startdatum geordnet sein:

```
class Project < ActiveRecord::Base
  has_many :iterations, :order => "start_date asc"
end
```

:counter_cache

Die Parent-Klasse einer 1:N-Relation besitzt die Methode *iterations.size* zur Ermittlung der Anzahl der assoziierten Child-Objekte. Jeder Aufruf dieser Methode resultiert in einer entsprechenden Datenbankabfrage. Dieser unnötige Datenbankzugriff lässt sich durch Verwendung des Parameters *counter_cache* vermeiden:

```
class Iteration < ActiveRecord::Base
  belongs_to :project, :counter_cache => true
end
```

Hat *:counter_cache* den Wert *true*, wird die Anzahl der assoziierten Child-Objekte, d.h. in diesem Fall die Anzahl der Iterationen eines Projekts, gecached. Das Ein- und Ausschalten dieser Option findet zwar in der Child-Klasse statt, der eigentliche Cache befindet sich aber in der Datenbanktabelle der Parent-Klasse:

```
class AddIterationsCountToProjects < ActiveRecord::Migration
  def self.up
    add_column :projects, :iterations_count, :integer,
              :null => false, :default => 0
  end

  def self.down
    remove_column :projects, :iterations_count
  end
end
```

Für die Definition des Caches gilt die Namenskonvention, dass das Cache-Feld den Namen der Assoziation, gefolgt von *_count*, haben muss. Der Default-Wert des Felds muss 0 sein.

Der Cache wird beim Hinzufügen bzw. Entfernen von Child-Objekten automatisch aktualisiert. Beim Zugriff auf die *size*-Methode der Child-Liste des Parent-Objekts erfolgt nun keine Datenbankabfrage mehr, stattdessen liefert die Methode den aktuellen Wert des Counter-Caches.

Hinweis: Iterationen können nicht nur einem Projekt, sondern ein Projekt kann auch einer Iteration zugewiesen werden. In diesem Fall wird der Cache nicht aktualisiert. Deshalb gilt: Wenn Sie mit einem Counter-Cache arbeiten, sollten Sie die Relation nur über die Parent-Klasse pflegen.

Tabelle 4.3: 1:N-Assoziation: Deklaration und erzeugte Methoden

1:N-Assoziation	Parent: Project	Child-Klasse: Iteration
Deklaration	has_many()	belongs_to()
erzeugte Methoden	iterations() iterations= iterations<<() iterations.delete() iterations.empty?() iterations.clear() iterations.size() iterations.find() iterations.build() iterations.create()	project() project= project.nil?() create_project() build_project()

Selbstreferenzierung

Modelle können sich selber referenzieren. Beispielsweise kann ein Task mehrere Subtasks besitzen, die selber wiederum vom Typ *Task* sind. Mit Active Record wird das wie folgt modelliert:

```
class Task < ActiveRecord::Base
  has_many :subtasks, :class_name => "Task",
    :foreign_key => "parent_id"
end

task = Task.new(:name => "Auto waschen")
task.subtasks << Task.new(:name => "Felgen putzen")
task.save
task.subtasks.map(&:name)

=> ["Felgen putzen"]
```

1:N-Assoziation: Deklaration und erzeugte Methoden

Die Tabelle 4.3 auf der vorherigen Seite fasst die Deklaration von 1:N-Assoziationen sowie die von der Deklaration erzeugten Methoden zusammen.

4.9.4 N:M-Beziehungen: *has_and_belongs_to_many*

In einer N:M-Beziehung steht ein Parent-Objekt in Relation zu 0-N Child-Objekten. Gleichzeitig steht ein Child-Objekt in Relation zu 0-N Parent-Objekten. Ein Eltern-Teil kann also mehrere Kinder, und ein Kind kann mehrere Eltern haben. Ein gutes Beispiel für diese Art der Assoziation ist die Beziehung zwischen einem Projekt und dessen Mitarbeitern: Ein Projekt hat mehrere Mitarbeiter, ein Mitarbeiter kann aber auch in mehreren Projekten arbeiten. Abbildung 4.3 veranschaulicht diesen Sachverhalt.

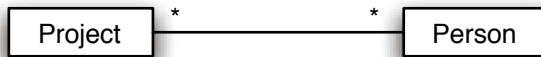


Abbildung 4.3: Mitarbeiter arbeiten an mehreren Projekten

N:M-Beziehungen werden auf Datenbankebene durch Join-Tabellen modelliert. Eine Join-Tabelle stellt die Verbindung zwischen Objekten her, indem sie die Primärschlüssel von je zwei Objekten aufeinander abbildet. Active Record nimmt als Namen der Join-Tabelle die verketteten Namen der assoziierten Tabellen in alphabetischer Reihenfolge an:

```

class CreatePeopleProjects < ActiveRecord::Migration
  def self.up
    create_table :people_projects do |t|
      t.integer :project_id, :person_id
    end
  end

  def self.down
    drop_table :people_projects
  end
end
  
```

Auf Modellebene werden N:M-Beziehungen durch die Assoziationsmethode *has_and_belongs_to_many* modelliert. Sie muss in beide Klassen der Assoziation eingefügt werden:

```

class Person < ActiveRecord::Base
  has_and_belongs_to_many :projects
end

class Project < ActiveRecord::Base
  has_and_belongs_to_many :members, :class_name => "Person"
end
  
```

Von „has_and_belongs_to_many“ erzeugte Methoden

Die von *has_and_belongs_to_many* erzeugten Methoden sind für beide Klassen der Relation identisch. Wir beschreiben exemplarisch die für die Klasse *Project* erzeugten Methoden.

members

Liefert ein Array der assoziierten Personenobjekte.

members<<(Person)

Fügt eine Person ans Ende der Personenliste des Projekts und speichert die Assoziation in der Join-Tabelle:

```
project = Project.create(:name => "OnTrack")
person = Person.create(:firstname => "Ralf")
project.members << person
project.members.map(&:firstname)
=> ["Ralf"]
```

members.push_with_attributes(Person, join_attributes)

Fügt eine Person ans Ende der Personenliste und speichert die Assoziation in der Join-Tabelle. Die Hash *join_attributes* enthält zusätzliche Werte, die für die Assoziation in der Join-Tabelle gespeichert werden. Die Schlüssel der Hash sind dabei Spaltennamen, d.h. die Join-Tabelle muss neben den IDs der assoziierten Objekte über genau diese zusätzlichen Felder verfügen.

Diese Methode sollten Sie verwenden, wenn Sie über die eigentliche Assoziation hinausgehende Attribute benötigen. Diese Attribute sind in beiden Objekten des Mappings über gleichnamige Getter und Setter verfügbar.

members.delete(Person, ...)

Löscht eine oder mehrere Personen aus der Liste:

```
project.delete person
assert_equal 0, project.members.size
```

members=(people)

Fügt die Liste *people* der Mitgliederliste zu und speichert die Assoziationen in der Join-Tabelle. Eine Person wird nur hinzugefügt, wenn sie noch nicht in der Mitgliederliste des Projekts enthalten ist. Im folgenden Beispiel wird *ralf* zweimal und *thomas* einmal hinzugefügt. Trotzdem enthält die Liste nur zwei Einträge:

```
project = Project.create(:name => "OnTrack")
ralf = Person.create(:firstname => "Ralf")
project.members << ralf
thomas = Person.create(:firstname => "Thomas")
project.members = [ralf, thomas]
assert_equal 2, project.members.size
```

member_ids=(ids)

Ersetzt die Mitgliederliste durch Personen, deren IDs den IDs der übergebenen Liste entsprechen:

```

project = Project.create(:name => "OnTrack")
ralf = Person.create(:firstname => "Ralf")
thomas = Person.create(:firstname => "Thomas")
astrid = Person.create(:firstname => "Astrid")
project.members = [ralf, thomas]
project.member_ids = [astrid.id]
assert_equal 1, project.members.size
assert_equal "Astrid", project.members[0].firstname

```

member_ids()

Liefert die Personen in der Mitgliederliste in Form der IDs.

members.clear

Löscht alle Personen aus der Mitgliederliste. Die Personenobjekte bleiben dabei erhalten und persistent.

members.empty?

Liefert *true*, wenn die Mitgliederliste leer ist, andernfalls *false*.

members.size

Liefert die Anzahl der Mitglieder des Projekts.

members.find(id)

Sucht in den assoziierten Personen nach dem Objekt mit der gegebenen ID:

```

project = Project.create(:name => "OnTrack")
ralf = Person.create(:firstname => "Ralf")
project.members << ralf
assert_equal ralf, project.members.find(ralf.id)

```

Parameter der „has_and_belongs_to_many“-Deklaration

Wie bei den anderen Assoziationsmakros erwartet die Methode als einzigen Pflichtparameter den Namen der Assoziation. Dieser Abschnitt beschreibt die optionalen Parameter der Assoziation.

:class_name

Dient der expliziten Angabe des Klassennamens, wenn der Name der Assoziation nicht mit dem Namen der assoziierten Klassen übereinstimmt. Wir verwenden diesen Parameter bereits in unserer Project-Member-Assoziation:

```

class Project < ActiveRecord::Base
  has_and_belongs_to_many :members, :class_name => "Person"
end

```

:join_table

Dient der expliziten Angabe der Join-Tabelle, sofern deren Name nicht den aneinandergefügt Namen der assoziierten Klassen entspricht.

:foreign_key

Standardmäßig nimmt Active Record als Namen für Fremdschlüssel den Namen der Modellklasse, gefolgt von *_id*, an. D.h., der Fremdschlüssel für *Project* heißt in der Join-Tabelle *project_id*. Der Parameter *foreign_key* erlaubt die explizite Angabe des Fremdschlüssels.

:association_foreign_key

Erlaubt die explizite Angabe des Fremdschlüssels der assoziierten Klassen. In den bisherigen Beispielen haben wir die Klasse *Person* assoziiert, d.h. Active Record hat als Fremdschlüssel in der Join-Tabelle den Namen *person_id* angenommen. Heißt der Schlüssel z.B. *member_id*, dann müssen *Project* und *Person* wie folgt deklariert werden:

```
class Project < ActiveRecord::Base
  has_and_belongs_to_many :members, :class_name => "Person",
                                   :association_foreign_key => "member_id"
end

class Person < ActiveRecord::Base
  has_and_belongs_to_many :projects, :foreign_key => "member_id"
end
```

:conditions

Bedingungen, unter denen assoziierte Personenobjekte in die Mitgliederliste aufgenommen werden. Die Bedingung wird im Format eines where-Statements angegeben. Beispielsweise stellt die folgende Deklaration sicher, dass nur gültige Personen in die Mitgliederliste geladen werden:

```
class Project < ActiveRecord::Base
  has_and_belongs_to_many :members,
                           :class_name => "Person",
                           :conditions => "valid = 1"
end
```

:order

Gibt die Reihenfolge der assoziierten Objekte an. Z.B. sollen Personen alphabetisch geordnet sein:

```
class Project < ActiveRecord::Base
  has_and_belongs_to_many :members,
                           :class_name => "Person", :order => "lastname"
end
```

:uniq

Boolescher Wert, der angibt, ob Duplikate in der Liste assoziierter Objekte ignoriert werden sollen.

:finder_sql

Alternatives SQL-Statement zum Laden der Assoziation.

:delete_sql

Alternatives SQL-Statement zum Löschen von Assoziationen, d.h. zum Löschen von Einträgen aus der Join-Tabelle.

insert_sql

Alternatives SQL-Statement zum Einfügen von Einträgen in die Join-Tabelle.

N:M-Assoziation: Deklaration und erzeugte Methoden

Tabelle 4.4 fasst die Deklaration von N:M-Assoziationen sowie die von der Deklaration erzeugten Methoden zusammen.

Tabelle 4.4: N:M-Assoziation: Deklaration und erzeugte Methoden

N:M-Assoziation	N-Klasse: Project	M-Klasse: Person
Deklaration	<code>has_and_belongs_to_many :members, :class_name => 'Person'</code>	<code>has_and_belongs_to_many :projects</code>
erzeugte Methoden	<code>members()</code> <code>members<<()</code> <code>members.push_with_attributes()</code> <code>members.delete()</code> <code>members=()</code> <code>member_ids=()</code> <code>member_ids()</code> <code>members.clear()</code> <code>members.empty?()</code> <code>members.size()</code> <code>members.find()</code>	<code>projects()</code> <code>projects<<()</code> <code>projects.push_with_attributes()</code> <code>projects.delete()</code> <code>projects=()</code> <code>project_ids=()</code> <code>project_ids()</code> <code>project.clear()</code> <code>project.empty?()</code> <code>project.size()</code> <code>project.find()</code>

4.9.5 Polymorphe Assoziationen: has_many – belongs_to

Polymorphe Assoziationen sind seit Rails 1.1 verfügbar und ermöglichen die Modellierung von 1:N-Beziehungen, in denen die 1-Seite der Assoziation polymorph ist, d.h. unterschiedliche Formen annehmen kann.

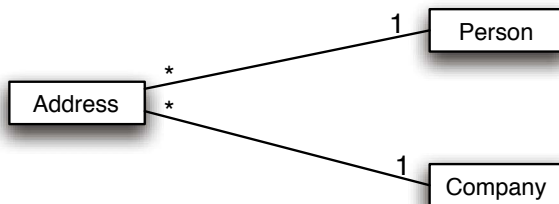


Abbildung 4.4: Polymorphe Assoziation

Das klingt komplizierter, als es ist: Angenommen, Sie wollen das Ontrack-Domain-Modell um die Klassen *Address* und *Company* erweitern, wobei eine *Company* beliebig viele Adressen haben kann. Das Gleiche gilt für die bereits existierende Klasse *Person*, die analog zu *Company* eine 1:N-Beziehung zur Klasse *Address* hat (siehe Abbildung 4.4).

Auf der 1-Seite der Assoziation befinden sich zwei unterschiedliche Klassen: *Person* und *Company*. Eine klassische *has_many*-Assoziation würde hier versagen, da die Assoziation auf beiden Seiten einen festen Typ erwartet, z.B.:

```
class Person < ActiveRecord::Base
  has_many :addresses
end

class Address < ActiveRecord::Base
  belongs_to :person
end
```

Mit Hilfe polymorpher Assoziationen lässt sich die beschriebene Beziehung sehr einfach modellieren. Wir beginnen mit der Erstellung der Tabellen für *Person*, *Address* und *Company*:

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.string :firstname
      t.integer :company_id
    end
  end

  def self.down
    drop_table :people
  end
end

class CreateAddresses < ActiveRecord::Migration
  def self.up
    create_table :addresses do |t|
      t.string :street, :city, :country
      t.integer :addressable_id
      t.string :addressable_type
    end
  end

  def self.down
    drop_table :addresses
  end
end

class CreateCompanies < ActiveRecord::Migration
  def self.up
    create_table :companies do |t|
```



```

        t.string :name
      end
    end

    def self.down
      drop_table :companies
    end
  end
end

```

Zunächst fällt auf, dass die Tabelle *addresses* statt eines Fremdschlüssels *person_id* bzw. *company_id* die beiden Felder *addressable_id* und *addressable_type* enthält. Die beiden Felder definieren zusammen einen Fremdschlüssel auf ein „adressierbares Ding“, d.h. auf eine Entität, die Adressen besitzen kann. In unserem Beispiel sind dies die Modellklassen *Person* und *Company*, die wir um entsprechende *has_many*-Deklarationen erweitern:

```

class Person < ActiveRecord::Base
  has_many :addresses, :as => :addressable
end

class Company < ActiveRecord::Base
  has_many :addresses, :as => :addressable
end

```

Die *has_many*-Deklaration hat einen neuen Parameter *:as*. Der Parameter bestimmt die Rolle des Modells innerhalb der Assoziation, d.h. *Person* und *Company* sind „adressierbare Dinge“. Was jetzt noch fehlt, ist die *belongs_to*-Deklaration auf der Adress-Seite der Relation:

```

class Address < ActiveRecord::Base
  belongs_to :addressable, :polymorphic => true
end

```

Auch auf dieser Seite der Assoziation gibt es einen neuen Parameter *:polymorphic*. Der Parameter stellt sicher, dass Active Record beim Lookup des assoziierten Objekts nicht nur den Fremdschlüssel *addressable_id* verwendet, sondern zusätzlich den Typ des assoziierten Objekts beachtet: *addressable_type*. D.h. Active Record sucht nicht in der Tabelle *addressables* (die gibt es nicht!) nach einem Objekt mit der Id *addressable_id*, sondern in der Tabelle mit dem Namen des pluralisierten Typs in *addressable_type*. Das folgende Beispiel verdeutlicht die Funktionsweise:

```

# Person mit Adresse
person_address = Address.create(:city => "Hamburg")
person = Person.create(:firstname => "Thomas")
person_address.addressable = person

puts "Addressable Class: #{person_address.addressable.class}"
puts "Person Name      : #{person_address.addressable.firstname}"

# Company mit Adresse
company_address = Address.create(:city => "London")

```

```

company = Company.create(:name => "b-simple")
company_address.addressable = company

puts "Addressable Class: #{company_address.addressable.class}"
puts "Company Name      : #{company_address.addressable.name}"

$ Addressable Class: Person
  Person Name       : Thomas
  Addressable Class: Company
  Company Name      : b-simple

```

Im ersten Teil des Beispiels legen wir eine Adresse an und assoziieren sie mit einer Person. Entsprechend speichert Active Record im Feld *addressable_type* den Namen der assoziierten Klasse *Person*. Im zweiten Teil des Beispiels handelt es sich bei dem assoziierten Objekt um eine Firma, d.h. eine Instanz der Klasse *Company*. Entsprechend enthält das Feld *addressable_type* der zweiten Adresse den Namen *Company*. Die von einer polymorphen *has_many_belongs_to*-Assoziation erzeugten Methoden entsprechen denen herkömmlicher *has_and_belongs_to_many*-Assoziationen (vgl. Abschnitt 4.9.3).

4.9.6 has_many :through

Seit Rails Version 1.1 ist eine neue Variante von N:M-Beziehungen verfügbar: *has_many :through* (im Folgenden kurz *:through* genannt). Während die in Abschnitt 4.9.4 beschriebenen N:M-Beziehungen (*has_many_and_belongs_to*) mit einer Join-Tabelle (ohne Primärschlüssel) auskommen, benötigen *:through*-Assoziationen neben der Join-Tabelle (mit Primärschlüssel) ein vollwertiges Active Record-Modell. Wir demonstrieren das Konzept am Beispiel der bereits bekannten N:M-Beziehung zwischen *Project* und *Person*. Als Erstes erzeugen wir die Join-Tabelle:

```

class CreateMemberships < ActiveRecord::Migration

  def self.up
    create_table :memberships do |t|
      t.integer :project_id, :person_id
    end
  end

  def self.down
    drop_table :memberships
  end
end

```

Das zugehörige Join-Modell *Membership* modelliert eine Beziehung zwischen einer Person und einem Projekt:

```

class Membership < ActiveRecord::Base
  belongs_to :project
  belongs_to :person
end

```

Was noch fehlt, sind die Erweiterungen der bereits bekannten Modellklassen *Project* und *Person* um die erforderlichen *has_many*-Assoziationen:

```
class Project < ActiveRecord::Base
  has_many :memberships
  has_many :people, :through => :memberships
end

class Person < ActiveRecord::Base
  has_many :memberships
  has_many :projects, :through => :memberships
end
```

Ein *Project* hat zwei *has_many*-Assoziationen: *memberships* und *people*. Die *memberships*-Assoziation enthält die Beziehungsobjekte, d.h. pro Projektmitglied eine *Membership*-Instanz. Die Mitglieder des Projekts sind in der *people*-Assoziation enthalten. Der neue Parameter *:through => :memberships* sorgt dafür, dass der Zugriff auf die Mitglieder eines Projekts „durch“ oder „über“ die *memberships*-Assoziation erfolgt.

Die *Person*-Klasse ist ähnlich aufgebaut: Der *:through*-Parameter sorgt dafür, dass sich die *projects*-Assoziation ihre *Project*-Objekte aus der *memberships*-Assoziation holt. Folgendes Beispiel demonstriert die Verwendung der neuen Beziehungen:

```
project = Project.create(:name => "OnTrack")
person = Person.create(:firstname => "Ralf")
Membership.create(:project => project, :person => person)

puts "Mitarbeiter: #{project.people[0].firstname}"
puts "Projekt      : #{person.projects[0].name}"

$ Mitarbeiter: Ralf
  Projekt      : OnTrack
```

Beachten Sie, dass das Beziehungsobjekt im obigen Beispiel explizit erzeugt wurde. Der Grund dafür ist, dass der von *has_many* und *has_and_belongs_to_many* bekannte *<<-*Operator zum Hinzufügen von Objekten für *through*-Assoziationen nicht funktioniert. Dies ist kein Fehler, sondern aufgrund von Active Record-Interna bewusst so implementiert.⁷

Sie fragen sich vielleicht, was denn der große Vorteil von *:through*-Beziehungen zu klassischen *has_and_belongs_to_many*-Beziehungen ist. Ein wichtiger Punkt ist sicherlich, dass *:through*-Assoziationen neben den zueinander in Beziehung stehenden Objekten weitere Attribute besitzen können. Denkbar wäre z.B. ein Rollenobjekt, welches die Rolle der Person im jeweiligen Projekt modelliert. Das geht zwar bei *has_and_belongs_to_many*-Beziehungen auch, allerdings ist die dafür benötigte Methode *push_with_attributes* mittlerweile veraltet, d.h. *deprecated* markiert.

⁷ Interessierte Leser finden unter <http://blog.hasmanythrough.com/articles/read/150> eine Erklärung für dieses Verhalten.

4.10 Aggregation

Aggregation bezeichnet eine besondere Art der Assoziation zwischen Objekten, in der ein Objekt „Teil“ eines anderen „ganzen“ Objekts ist.

In Rails werden Aggregationen mit Hilfe der *belongs_to*-Assoziation modelliert. Dies setzt voraus, dass sowohl das „ganze“ als auch das „Teil“-Objekt Active Record-Modelle sind. Dass dies nicht immer der Fall ist, zeigt folgendes Beispiel: Ein Benutzer hat eine Adresse. Adressdaten, wie Straße oder Ort, werden als zusätzliche Felder in der Tabelle *users* gespeichert:

```
class AddAddressToUsers < ActiveRecord::Migration
  def self.up
    add_column :people, :street, :string
    add_column :people, :city, :string
  end

  def self.down
    remove_column :people, :street
    remove_column :people, :city
  end
end
```

Im Sinne eines sauberen Objektmodells werden Adressen auf Modellebene als aggregierte Objekte der Klasse *User* modelliert:

Listing 4.1: Aggregierte Adressklasse

```
class Address
  attr_reader :street, :city

  def initialize(street, city)
    @street = street
    @city = city
  end
end
```

composed_of

Aggregationen, deren Teil-Objekte keine Active Record-Objekte sind, werden durch die Deklaration *composed_of* modelliert:

```
class User < ActiveRecord::Base
  composed_of :address, :class_name => Address,
    :mapping => [ %w(address_street street), %w(address_city city) ]
end
```

Der erste Parameter von *composed_of* gibt den Namen der Aggregation an. Der Parameter *:class_name* spezifiziert den Typ der aggregierten Klasse. Entspricht der Typ dem Namen der Aggregation (wie in diesem Fall), ist die Angabe von *:class_name* optional. Der Parameter *:mapping* spezifiziert eine Anzahl von Mapping-Arrays (Attribut, Konstruktorparameter), die jeweils ein Attribut der aggregierenden Klasse (User) auf einen Konstruktorparameter der aggregierten Klasse (Address) abbilden.

Im Beispiel wird also das Attribut *address.street* von *User* auf den Konstruktorparameter *street* von *Address* abgebildet sowie das Attribut *address.city* von *User* auf den Konstruktorparameter *city* von *Address*.

Die Deklaration erweitert die Klasse *User* um die Methoden *address* und *address=* für den Zugriff auf bzw. die Zuweisung von aggregierten *Address*-Objekten:

```
user = User.create
address = Address.new("Große Freiheit", "Hamburg")
user.address = address
user.reload
puts "Street: " + user.address.street
puts "City  : " + user.address.city
```

Die aggregierte Klasse benötigt einen Konstruktor, der die zugehörigen Felder in der Form akzeptiert, in der sie in der Tabelle definiert wurden (vgl. Listing 4.1). Außerdem benötigt die Klasse für jedes zugehörige Tabellenfeld eine gleichnamige Getter-Methode.

Hinweis: Teil-Objekte im Sinne einer *composed_of*-Aggregation sind Value-Objekte, die nach ihrer Erzeugung nicht verändert werden sollten.

4.11 Vererbung

Ruby ist eine objektorientierte Programmiersprache. Ein wichtiges Konzept der Objektorientierung ist die Vererbung, auf die wir bei der Verwendung von Active Record keinesfalls verzichten müssen.

Um das Konzept zu demonstrieren, erweitern wir unsere Projektverwaltung um eine neue Modellklasse *Manager*. Die *Manager*-Klasse entspricht im Großen und Ganzen der bereits vorhandenen Klasse *Person*, unterscheidet sich jedoch in wenigen Details und muss entsprechend erweitert werden. Damit wir Daten und Funktionalität von *Person* nicht duplizieren, lassen wir *Manager* von *Person* erben (siehe Abbildung 4.5).

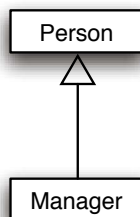


Abbildung 4.5: Manager sind auch nur Menschen.

Die spannende Frage, wie Active Record die Vererbungsrelation der Modellebene in der Datenbank abbildet,⁸ beantwortet (wie so häufig) ein von Martin Fowler beschriebenes Pattern: *Single Table Inheritance* (siehe [7]).

Single Table Inheritance (STI) verwendet für alle Klassen eines Vererbungszeigs dieselbe Datenbanktabelle. In unserem Beispiel werden sowohl Personen als auch Manager in der Tabelle *people* gespeichert. Als Konsequenz daraus resultiert, dass die verwendete Tabelle für jedes Attribut der zugehörigen Modellklassen ein entsprechendes Feld haben muss. Dies führt dazu, dass STI-Tabellen mehr Felder als notwendig und teilweise ungenutzte Felder enthalten, was aber in der Praxis kein wirkliches Problem ist.

Wie verhält es sich konkret mit unserem Beispiel? Manager-Objekte werden in der Tabelle *people* gespeichert. Demzufolge müssen wir diese Tabelle um zusätzliche Manager-Felder erweitern. Angenommen, ein Manager darf über ein Budget bis zu einer bestimmten Höhe verfügen, dann muss die Tabelle wie folgt erzeugt werden:

```
class AddBudgetToPeople < ActiveRecord::Migration
  def self.up
    add_column :people, :budget, :integer
  end

  def self.down
    remove_column :people, :budget
  end
end
```

Auf Modellebene wird die Vererbungsrelation zwischen Manager und Person ganz normal deklariert, d.h. *Person* erbt wie gehabt von *ActiveRecord::Base* und *Manager* erbt von *Person*:

```
class Person < ActiveRecord::Base
end

class Manager < Person
end
```

Die Instanzen beider Klassen werden jetzt in der Tabelle *people* gespeichert:

```
Person.create(:firstname => "Ralf")
Manager.create(:firstname => "Thomas", :budget => 1000)
```

```
mysql> select * from people;
+----+-----+-----+-----+
| id | firstname | budget | company_id |
+----+-----+-----+-----+
| 1  | Ralf      | NULL  | NULL       |
| 2  | Thomas    | 1000  | NULL       |
+----+-----+-----+-----+
2 rows in set (0.00 sec)
```

⁸ Ein klassisches Problem des OR-Mappings.

Allerdings haben wir noch ein Problem: Nachdem die Objekte einmal in der Tabelle gespeichert sind, kann Active Record nicht mehr erkennen, ob es sich bei den gespeicherten Daten um *Manager*- oder *Person*-Daten handelt. Die Erweiterung der Tabelle *people* um das Feld *type* (kann durch *Base.inheritance_column* überschrieben werden) löst dieses Problem und schaltet den eigentlichen STI-Mechanismus an:

```
class AddTypeToPeople < ActiveRecord::Migration
  def self.up
    add_column :people, :type, :string
  end

  def self.down
    remove_column :people, :type
  end
end
```

Active Record sorgt jetzt dafür, dass der Typ von abgeleiteten Klassen automatisch in die zugehörige Tabelle geschrieben wird:

```
Person.create(:firstname => "Ralf")
Manager.create(:firstname => "Thomas", :budget => 1000)
```

```
mysql> select * from people;
+----+-----+-----+-----+-----+
| id | firstname | budget | company_id | type      |
+----+-----+-----+-----+-----+
| 1  | Ralf      | NULL   | NULL       | NULL      |
| 2  | Thomas    | 1000   | NULL       | Manager    |
+----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Active Record nutzt die Typinformation beim Laden von Objekten, indem abhängig vom gespeicherten Typ die korrekten Instanzen erzeugt werden. Beispielsweise liefert *Person.find* ein gemischtes Array, bestehend aus *Person*- und *Manager*-Objekten:

```
persons = Person.find(:all)
puts persons[0].inspect
puts persons[1].inspect

$ #<Person:0x6777cc @attributes={"type"=>nil, "id"=>"1",
  "firstname"=>"Ralf", "company_id"=>nil, "budget"=>nil}>
  #<Manager:0x677628 @attributes={"type"=>"Manager",
    "id"=>"2", "firstname"=>"Thomas", "company_id"=>nil,
    "budget"=>"1000"}>
```

Während *Person.find* alle *Person*-Objekte einliest, lädt *Manager.find* nur noch *Manager*-Instanzen:

```
puts Manager.find(:all).inspect

$ [#<Manager:0x65ac58 @attributes={"type"=>"Manager",
```

```
"id"=>"2", "firstname"=>"Thomas", "company_id"=>nil,
"budget"=>"1000"}>]
```

4.12 Transaktionen

Wohl kaum eine Geschäftsanwendung kommt ohne Transaktionen aus. Eine Transaktion ist ein Block zusammengehöriger SQL-Statements, in dem entweder alle oder kein Statement ausgeführt werden.

Das klassische Anwendungsbeispiel für Transaktionen ist die Überweisung eines Betrags zwischen zwei Konten. Sowohl die Abbuchung als auch die Gutschrift müssen funktionieren. Gelingt stattdessen nur eine Abbuchung ohne anschließende Gutschrift, verschwindet der Betrag im Nirvana. Um dies zu verhindern, klammert man beide Aktionen in einer Transaktion.

Hinweis: Bevor wir in die Transaktionsmechanismen von Active Record einsteigen, noch ein kurzer, aber wichtiger Hinweis bezüglich Transaktionen unter MySQL: Damit Transaktionen unter MySQL funktionieren, müssen die beteiligten Tabellen vom Typ *InnoDB* sein. Damit die Beispiele dieses Abschnitts funktionieren, müssen die Tabellen *iterations* und *tasks* entsprechend geändert werden:

```
alter table iterations TYPE=InnoDB;
alter table tasks TYPE=InnoDB;
```

Ein Anwendungsbeispiel für Transaktionen in unserem Projektmanagement-System ist das Verschieben von Tasks zwischen Iterationen. Nehmen wir an, Sie stellen gegen Ende der ersten Iteration fest, dass die Zeit zur Fertigstellung von Task X nicht mehr ausreicht. Sie entscheiden, Task X in Iteration 2 zu verschieben, um so die erste Iteration termingerecht abschließen zu können. In diesem Fall ist es wichtig, dass sowohl das Entfernen aus Iteration 1 als auch das Hinzufügen zu Iteration 2 funktionieren. Klappt nur das Entfernen, verschwindet Task X wie ein abgebuchter Betrag im Nirvana. Die Lösung ist also auch hier die Klammerung des Löschens und des Hinzufügens in einer Transaktion.

Zentrale Transaktionsmethode von Active Record ist die Klassenmethode *transaction*, die jeder Modellklasse zur Verfügung steht. Die Methode erwartet einen Block mit den transaktional durchzuführenden Anweisungen. Wir implementieren die Verschiebefunktion für Tasks als neue Methode der Klasse *Iteration*:

```
class Iteration < ActiveRecord::Base
  def move_task(task, iteration)
    transaction do
      tasks.delete task
      iteration.tasks << task
    end
  end
end
```

Die Transaktion wird nur dann „committed“, wenn innerhalb des Blocks keine Exception geworfen wird.

Beim Hinzufügen des Tasks zur zweiten Iteration könnte etwas schiefgehen, z.B. das Enddatum des Tasks vor dem Startdatum der Iteration liegen, wodurch die Validierung der Ziel-Iteration fehlschlägt. Das wiederum würde dazu führen, dass die Ziel-Iteration nicht gespeichert wird, d.h. die `save`-Methode liefert `false`. Das folgende Codebeispiel stellt das beschriebene Szenario nach:

```
def test_move_task
  i1 = Iteration.new(:start_date => Date.new(2006, 6, 27))
  i1.tasks << t1 = Task.new(:due_date => Date.new(2006, 7, 26))
  i1.save

  i2 = Iteration.new(:start_date => Date.new(2006, 7, 27))
  i2.tasks << t2 = Task.new(:due_date => Date.new(2006, 7, 30))
  i2.save

  begin
    i1.move_task(t1, i2)
  rescue
  end

  assert_equal 1, i1.reload.tasks.length
  assert_equal 1, i2.reload.tasks.length
end
```

Task *t1* hat ein Zieldatum, das kleiner als das Startdatum von *i2* ist, sodass die Validierung von *i2* schiefgeht. Da wir `move_task` transaktional programmiert haben, können wir davon ausgehen, dass *t1* nicht verschoben wird, d.h. beide Iterationen nach Ausführung der `move_task`-Methode immer noch einen Task enthalten. Die Ausführung des Tests liefert aber folgendes Ergebnis:

```
1) Failure:
test_move_task(TransactionTest) [transaction_test.rb:27]:
<1> expected but was
<0>.
```

Iteration *i1* enthält keine Tasks mehr. Das Problem ist, dass die beteiligten Objekte implizit gespeichert werden. Z.B. speichert der in der Transaktion verwendete Aufruf `iteration.tasks << task` das Task-Objekt, um sich die zugehörige Iteration zu merken (Task hält den Fremdschlüssel auf die Iteration). Und jetzt kommen wir zum Kern des Problems: Bei der impliziten Speicherung ruft Rails die `save`-Methode auf, die einen Booleschen Wert liefert, aber keine Exception wirft. Da die Transaktion nur beim Auftreten einer Exception zurückgerollt wird, passiert in unserem Beispiel gar nichts, weil die Transaktion nichts von dem aufgetretenen Problem mitkriegt. Abhilfe schafft ein expliziter `save!`-Aufruf. Sie erinnern sich, die Methode `save!` wirft bei fehlschlagender Validierung eine Exception. Wir müssen die `move_task`-Methode also wie folgt ändern:

```
def move_task(task, iteration)
  transaction do
    tasks.delete task
    iteration.tasks << task
  end
end
```

```
iteration.save!  
end  
end
```

Nach der Änderung läuft der Unit Test fehlerfrei durch, d.h. jede Iteration enthält nach dem Aufruf der *move_task*-Methode immer noch einen Task. Allerdings enthält der Testcode im obigen Listing zwei *reload*-Aufrufe für die beteiligten Iterationen. Der Grund dafür ist, dass Active Record-Transaktionen zunächst nur Auswirkungen auf die Datenbankinhalte haben. Die beteiligten Modellinstanzen werden auf jeden Fall geändert, unabhängig davon, ob die Transaktion erfolgreich verlief oder nicht.

Dieses Verhalten lässt sich durch die Verwendung von Transaktionen auf Objektebene vermeiden. Dabei müssen dem *transaction*-Aufruf alle an der Transaktion beteiligten Objekte explizit übergeben werden:

```
def move_task(task, iteration)  
  transaction(self, iteration) do  
    self.tasks.delete task  
    iteration.tasks << task  
    iteration.save!  
  end  
end
```

So geändert, führt Active Record Buch über die Änderungen der beteiligten Objekte, und die im Test verwendeten *reload*-Aufrufe werden nicht mehr benötigt.

Active Record nutzt den Transaktions-Mechanismus im Übrigen auch für sich selbst. Wie Sie wissen, werden Child-Objekte automatisch gespeichert, wenn das Parent-Objekt gespeichert wird. Active Record sorgt dafür, dass dies automatisch innerhalb einer Transaktion geschieht, d.h. geht das Speichern des Child-Objekts schief, dann wird auch das Parent-Objekt nicht gespeichert. Das Gleiche gilt für das Löschen von Objekten.

4.13 Von Bäumen und Listen

Bäume und Listen sind ein klassisches Werkzeug der Datenorganisation und deshalb in vielen Softwaresystemen zu finden. Beispielsweise enthält unser Projektmanagement-System Listen von Tasks oder baumartige Hierarchien von Projektmitgliedern.

Hinweis zu Rails 2.0: Sämtliche *acts_as*-Modellhelper wurden in Rails 2.0 aus dem Framework entfernt und in Plugins ausgelagert. Sie installieren die in diesem Abschnitt beschriebenen Helper *acts_as_list* und *acts_as_tree* wie folgt:

```
script/plugin install  
  http://svn.rubyonrails.org/rails/plugins/acts_as_list  
  
script/plugin install  
  http://svn.rubyonrails.org/rails/plugins/acts_as_tree
```

4.13.1 `acts_as_list`

Die `acts_as_list`-Deklaration organisiert die Child-Objekte einer 1:N-Relation als Objektliste mit fester Reihenfolge. Die `acts_as_list`-Deklaration ist sinnvoll, wenn die Child-Objekte eine bestimmte Ordnung aufweisen müssen, aber kein explizites Kriterium zum Herstellen dieser Ordnung mehr existiert. Beispiel: Unser Projektmanagement verwaltet Listen von Tasks, sortiert nach Priorität:

- T2: Projekte erfassen, Prio 1.
- T1: Infrastruktur aufsetzen, Prio 1.
- T3: Iterationen erfassen, Prio 1.

Alle Tasks der Liste haben eine Priorität von 1, d.h. sie sind gemäß unserer Sortierregelung zunächst gleichwertig. Dennoch ist es wichtig, dass T1 vor T2 und T2 vor T3 ausgeführt wird, wofür uns jedoch kein explizites Sortierkriterium zur Verfügung steht. Unser System benötigt eine Funktion, die es dem Anwender erlaubt, Tasks zu sortieren (siehe Abschnitt *Drag and Drop* im Kapitel 10). Weiterhin wird ein Mechanismus benötigt, der die vom Benutzer vorgegebene Reihenfolge aufrechterhält, und genau diese Funktion bietet `acts_as_list`. Für die Nutzung von `acts_as_list` sind drei Schritte notwendig:

1. Datenbanktabelle des Childs (Task) um Feld *position* erweitern.
2. `has_many`-Deklaration des Parents (Iteration) um den Parameter `:order` ergänzen.
3. Child-Klasse um eine `acts_as_list`-Deklaration erweitern.

Intern verwaltet Active Record die Position eines Child-Objekts mit Hilfe des Attributs *position*, das entsprechend in der Datenbanktabelle des Child-Objekts vorhanden sein muss:

```
class CreateTasks < ActiveRecord::Migration

  def self.up
    create_table :tasks do |t|
      t.string :name
      t.integer :priority, :iteration_id, :position
    end
  end

  def self.down
    drop_table :tasks
  end

end
```

Damit die Position der Objekte beim Laden berücksichtigt wird, muss die *has_many*-Deklaration des Parents um den *:order*-Parameter mit Verweis auf das *position*-Feld erweitert werden:

```
class Iteration < ActiveRecord::Base
  has_many :tasks, :order => :position
end
```

Abschließend muss die Child-Klasse um die *acts_as_list*-Deklaration erweitert werden. Der Parameter *:scope* gibt an, über welchen Bereich sich die Liste erstreckt. Für Tasks ist es z.B. nur wichtig, dass die Tasks bezogen auf eine Iteration in der richtigen Reihenfolge auftreten.

```
class Task < ActiveRecord::Base
  belongs_to :iteration
  acts_as_list :scope => :iteration_id
end
```

Für die so erweiterten Iterationen und Tasks stellt Active Record jetzt intern sicher, dass die Tasks in der vom Benutzer vorgegebenen Reihenfolge bleiben. Da wir noch keinen View zum Sortieren von Tasks haben, demonstrieren wir das Prinzip am Beispiel eines kleinen Ruby-Programms:

```
iteration = Iteration.new(:name => "I1")
iteration.tasks.create(:name => "Task 1: Projekt erfassen",
                     :priority => 1)
iteration.tasks.create(:name => "Task 2: Infrastruktur aufsetzen",
                     :priority => 1)
iteration.tasks.create(:name => "Task 3: Iteration erfassen",
                     :priority => 1)

iteration.save
puts iteration.tasks.map{ |task| task.name }.join(", ")
iteration.tasks[1].move_higher
iteration.reload
puts iteration.tasks.map{ |task| task.name }.join(", ")

$ Task 1: Projekt erfassen, Task 2: Infrastruktur aufsetzen,
  Task 3: Iteration erfassen
$ Task 2: Infrastruktur aufsetzen, Task 1: Projekt erfassen,
  Task 3: Iteration erfassen
```

Das Beispielprogramm fügt die neuen Tasks bewusst in falscher Reihenfolge ein: Die Erfassung von Projekten (Task 1) ist erst nach der Bereitstellung einer Infrastruktur (Task 2) möglich. Entsprechend wird Task 2 nach dem Speichern der Iteration um eine Position nach vorne geschoben (*move_higher*). Anschließend wird die Iteration neu geladen und die Taskliste ausgegeben. Beim Verschieben des Tasks setzt Active Record den Wert des *positions*-Felds automatisch so, dass beim Neuladen der Iteration die manipulierte Reihenfolge erhalten bleibt. Das Neuladen ist im Übrigen zwingend erforderlich, da die Iteration selber nichts von der Verschiebeaktion mitbekommt.

4.13.2 acts_as_tree

Die *acts_as_tree*-Deklaration organisiert die Child-Objekte einer 1:N-Assoziation in einer Baumstruktur. Baumstrukturen sind für die Modellierung von Organigrammen hilfreich, wie z.B. die Mitgliederhierarchie eines Projekts: Ganz oben in der Hierarchie steht der Projektleiter, darunter folgen die Teilprojektleiter, denen wiederum die einzelnen Entwickler untergeordnet sind (siehe Abbildung 4.6).

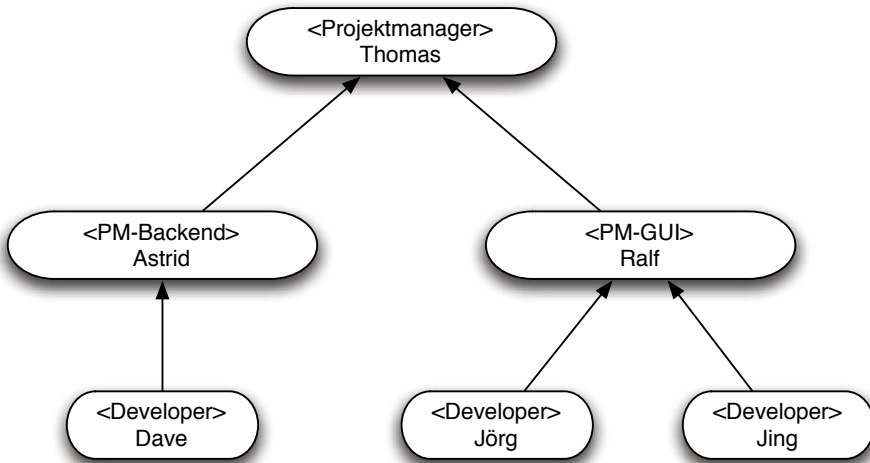


Abbildung 4.6: Projekt-Organigramm

Für die interne Organisation der Baumstruktur verwendet Active Record das Attribut *parent_id* des Child-Objekts, dessen Datenbanktabelle ein entsprechendes Integer-Feld haben muss:

```

class AddParentIdToPeople < ActiveRecord::Migration
  def self.up
    add_column :people, :parent_id, :integer
  end

  def self.down
    remove_column :people, :parent_id
  end
end

```

Das Feld *parent_id* darf *nil* sein, da das Wurzelobjekt des Baums kein Elternobjekt besitzt. Die Child-Klasse *Person* wird um die *acts_as_tree*-Deklaration erweitert.

```

class Person < ActiveRecord::Base
  has_and_belongs_to_many :projects
  acts_as_tree :foreign_key => "parent_id", :order => "firstname"
end

```

Der Parameter *:foreign_key* ist optional und muss nur angegeben werden, wenn für die Referenz auf das Parent-Objekt im Baum ein anderer Name als *parent_id*

verwendet wird. Der optionale Parameter `:order` gibt die Reihenfolge der Child-Objekte eines Zweigs vor. Das folgende Codebeispiel demonstriert die *acts_as_tree*-Deklaration in ihrer Anwendung:

```
project = Project.create(:name => "OnTrack")
pm = Person.create(:firstname => "Thomas")
project.members << pm

pm_gui = Person.create(:firstname => "Ralf", :parent => pm)
project.members << pm_gui
jing = Person.create(:firstname => "Jing", :parent => pm_gui)
project.members << jing
joerg = Person.create(:firstname => "Jörg", :parent => pm_gui)
project.members << joerg

pm_backend = Person.create(:firstname => "Astrid", :parent => pm)
project.members << pm_backend
dave = Person.create(:firstname => "Dave", :parent => pm_backend)
project.members << dave
project.reload

project.members.each do |person|
  print_person(person) if person.parent.nil?
end
```

Im Beispiel wird die Baumstruktur aus Abbildung 4.6 nachgebildet. Die Ausgabe der Projektmitglieder am Ende des Beispiels zeigt, dass sich die Mitglieder auch nach dem Neuladen des Projekts in der gewünschten Hierarchie befinden:

```
Thomas
--> Astrid
-----> Dave
--> Ralf
-----> Jörg
-----> Jing
```

Der Baum ist deutlich erkennbar: Astrid und Ralf arbeiten für Thomas, Dave arbeitet für Astrid, und Jörg und Jing arbeiten für Ralf. Der Vollständigkeit halber hier noch die Methode zur Ausgabe des Personenbaums:

```
def print_person(person, level = 4)
  puts person.firstname if person.parent.nil?
  person.children.each do |child|
    puts "> ".rjust(level, "--") + child.firstname
    print_person(child, level + 3)
  end
end
```

4.14 Validierung

Ein weiterer Bestandteil von Active Record ist das Validierungs-Framework. „Validierung“ bedeutet die Überprüfung der Daten eines Modells auf Gültigkeit. Sie findet auf Modellebene statt, wobei uns zwei Varianten zur Verfügung stehen:

- Validierungs-Klassenmethoden
- Die Methode *validate*

Wir fangen mit der Beschreibung der Methode *validate* an, da hierdurch die grundsätzliche Funktionsweise des Validierungsmechanismus deutlich wird. Die Methode ist in einer Active Record-Basisklasse definiert und kann überschrieben werden:

```
class Project < ActiveRecord::Base
  protected
  def validate
    if name.blank?
      errors.add("name", "Name ist nicht angegeben")
    end
  end
end
```

Die Methode *validate* überprüft die Attribute eines Modells auf Gültigkeit. In diesem Fall wird geprüft, ob das Projekt einen gültigen Namen besitzt. Wenn nicht, wird der vom Getter *errors* gelieferten Fehlerliste *errors* eine entsprechende Meldung hinzugefügt.

Die Methode *validate* wird von Active Record automatisch vor jedem Speichern einer Modellinstanz aufgerufen.⁹ *save* liefert einen Booleschen Wert, der anzeigt, ob die Instanz erfolgreich gespeichert werden konnte. Schlägt die Validierung fehl, wird das Modell nicht gespeichert, und *save* liefert *false*:

```
project = Project.new
puts "project.save => " + project.save.to_s
$ project.save => false
```

Das Objekt *errors* ist mehr als eine reine Fehlerliste. Die Klasse des Objekts *ActiveRecord::Errors* bietet neben der Methode *add* eine Reihe weiterer Methoden für die komfortable Verwaltung der Fehler eines Modells.

Beispielsweise können Sie mit der Methode *on* prüfen, ob für ein bestimmtes Attribut eine Fehlermeldung vorliegt:

```
puts project.errors.on(:name)
$ "Name" ist nicht angegeben.
```

⁹ Die automatische Validierung kann durch Verwendung von *save_with_validation(false)* unterbunden werden.

Wir wollen die Beschreibung der einzelnen Methoden der Klasse *Errors* an dieser Stelle nicht wiederholen und verweisen stattdessen auf die Rails-API-Dokumentation ([14]).

Neben *validate* definiert *ActiveRecord::Base* noch die beiden Methoden *validate_on_create* und *validate_on_update*. Je nachdem, ob man ein neues oder ein vorhandenes Objekt speichert, wird eine dieser beiden Methoden zusätzlich zu *validate* aufgerufen. Überschreiben Sie eine oder beide dieser Methoden, wenn die Validierung neuer und vorhandener Objekte unterschiedlich ist.

4.14.1 Validierungs-Klassenmethoden

Im vorigen Abschnitt haben Sie die manuelle, d.h. die „von Hand“ programmierte Validierung kennengelernt. Ziel dieses Vorgehens war die Schaffung eines Grundverständnisses für die Funktionsweise der Active Record-Validierung. In diesem Abschnitt lernen Sie die Luxusversion der Validierung kennen: die Nutzung der Active Record-Klassenmethoden zur Validierung.

Validierungs-Klassenmethoden sind Methoden, die in der Deklaration von Modellklassen verwendet werden. Im Regelfall müssen Sie also nicht mehr die *validate*-Methode überschreiben, sondern erweitern stattdessen die Deklarationen Ihrer Modelle um die Methodenaufrufe, die das gewünschte Validierungsverhalten implementieren. Wollen Sie z.B. sicherstellen, dass ein Attribut nicht leer ist, können Sie dafür die Methode *validates_presence_of* verwenden. Die Methode akzeptiert ein oder mehrere zu überprüfende Attribute in Symbolschreibweise:

```
class Project < ActiveRecord::Base
  validates_presence_of :name, :description
end
```

Als Standardfehlermeldung verwendet die Methode den Text „can’t be blank“. Dieser Text kann durch Verwendung des optionalen Attributs *:message* überschrieben werden:

```
validates_presence_of :name, :description,
                      :message => "darf nicht leer sein"
```

Der Parameter *:on* steuert, für welche Aktion die Validierung aktiv ist. Mögliche Werte für *:on* sind *:save* (default), *:create* und *:update*. Soll die Validierung beispielsweise nur beim Anlegen neuer Projekte aktiv sein, muss der Validierungscode folgendermaßen geändert werden:

```
validates_presence_of :name, :description,
                      :message => "darf nicht leer sein"
                      :on => :create
```

Alle nachfolgend beschriebenen Validierungsmethoden akzeptieren gleichermaßen die Parameter *:message* und *:on*, weshalb wir diese Parameter in den Beschreibungen der einzelnen Methoden nicht mehr explizit erwähnen.

Weitere Validierungsmethoden

Active Record definiert eine ganze Reihe weiterer Validierungsmethoden, die wir im Folgenden genauer vorstellen.

validates_acceptance_of

Jeder von uns kennt die Situation, dass wir beim Einrichten eines neuen Zugangs für einen Web-Shop eine Checkbox ankreuzen müssen, womit wir bestätigen, dass wir die Nutzungsbedingungen des Shops akzeptieren. Mit Hilfe der Methode *validates_acceptance_of* können wir sicherstellen, dass Checkboxes für Nutzungsbedingungen (oder Ähnliches) vom Benutzer selektiert werden:

```
class User < ActiveRecord::Base
  validates_acceptance_of :terms_of_service
end
```

Das Speichern funktioniert jetzt nur, wenn der Benutzer die entsprechende Checkbox selektiert. Hier das Gegenbeispiel:

```
user = User.new(:terms_of_service => false)
assert !user.save
```

validates_associated

Wenn ein Parent-Objekt gespeichert wird, validiert Active Record automatisch dessen assoziierte Child-Objekte. Der folgende *save*-Aufruf schlägt fehl, weil die assoziierte Iteration keinen Namen hat:

```
project = Project.new(:name => "OnTrack",
                     :description => "Easy Projectmanagement")
iteration = Iteration.new
project.iterations << iteration
assert !project.save
```

Allerdings wird das Attribut *errors* des Projekts nicht um eine explizite Fehlermeldung erweitert, sodass der Anwender nicht auf den Fehler hingewiesen werden kann. Hier kommt die Methode *validates_associated* ins Spiel:

```
class Project < ActiveRecord::Base
  validates_associated :iterations,
                      :message => "Iteration ohne Namen"
end
```

Der Aufruf von *validates_associated* bewirkt, dass das *errors*-Objekt von *Project* um eine Fehlermeldung ergänzt wird, wenn die Validierung einer der assoziierten Iterationen fehlschlägt.

validates_confirmation_of

Validiert, ob zwei Felder (z.B. Passwörter) denselben Inhalt haben. Dabei muss das Modell nur eines der beiden Felder wirklich besitzen. Das andere Feld dient als virtuelles Bestätigungsfeld. Beispiel:

```
class User < ActiveRecord::Base
  validates_confirmation_of :password
end

<%= password_field "person", "password" %>
<%= password_field "person", "password_confirmation" %>
```

Das im View verwendete Attribut *password_confirmation* ist rein virtuell, wird also nicht gespeichert und nur für die Überprüfung auf Falscheingabe benötigt. Wichtig ist die Einhaltung der Namenskonvention, d.h. das virtuelle Attribut hat den Namen des realen Attributs, gefolgt von *_confirmation*.

validates_each

Die Methode führt einen Block für das oder die angegebenen Attribute aus. Der folgende Code sorgt dafür, dass keine vergebenen Projektnamen verwendet werden:

```
class Project < ActiveRecord::Base
  validates_each :name do |record, attr, value|
    if value == "Toll Collect"
      record.errors.add attr, "Name bereits vergeben"
    end
  end
end
```

validates_exclusion_of

Stellt sicher, dass das Attribut nicht in der angegebenen Wertemenge enthalten ist. Folgender Code stellt sicher, dass keine reservierten Benutzernamen vergeben werden:

```
class User < ActiveRecord::Base
  validates_exclusion_of :name, :in => %w(root ralf thomas)
end
```

validates_format_of

Die Methode überprüft mit Hilfe eines regulären Ausdrucks das Format des angegebenen Attributs. Der folgende Code stellt sicher, dass die E-Mail-Adresse eines Benutzers ein gültiges Format besitzt. Der Parameter *:with* spezifiziert den regulären Ausdruck:

```
class User < ActiveRecord::Base
  :validates_format_of :email,
    :with => /^[^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$/i,
    :on => :create
end
```

validates_inclusion_of

Prüft, ob das Attribut in der durch den Parameter *:in* angegebenen Menge enthalten ist. Der folgende Code validiert, ob das Attribut *gender* (Geschlecht) einen der Werte *m* (Mann) oder *f* (Frau) besitzt:

```
class User < ActiveRecord::Base
  validates_inclusion_of :gender, :in => %w( m f ),
    :message=>"Was sind Sie dann?"
end
```

validates_length_of

Validiert die Länge des Attributs. Beispielsweise müssen Benutzernamen eine Mindestlänge von 4 Zeichen haben:

```
class User < ActiveRecord::Base
  validates_length_of :name, :minimum => 4
end
```

Benutzernamen dürfen aber auch nicht länger als 8 Zeichen sein:

```
class User < ActiveRecord::Base
  validates_length_of :name, :minimum => 4
  validates_length_of :name, :maximum => 8
end
```

Einfacher wird es, wenn Sie für eine Bereichsvalidierung den *:within*-Parameter¹⁰ verwenden:

```
class User < ActiveRecord::Base
  validates_length_of :name, :within => 4..8
end
```

Die Validierung von exakten Längen erfolgt durch Verwendung des Parameters *:is*:

```
class User < ActiveRecord::Base
  validates_length_of :name, :is => 6
end
```

Die Standardfehlermeldungen für zu kurze, zu lange oder inexakte Attributwerte können durch die Parameter *:too_short*, *:too_long* und *:wrong_length* überschrieben werden.

```
class User < ActiveRecord::Base
  validates_length_of :name, :within => 4..8,
    :too_short => "ist zu kurz",
    :too_long => "ist zu lang"
end

user = User.new
user.update_attributes({:name => "Ral"})
puts "'#{user.name}' #{user.errors.on(:name)}"

user.update_attributes({:name => "Ralf Wirdemann"})
puts "'#{user.name}' #{user.errors.on(:name)}"

$ 'Ral' ist zu kurz
  'Ralf Wirdemann' ist zu lang
```

¹⁰oder dessen Alias *:in*

Die *puts*-Aufrufe des obigen Codeausschnitts verwenden die Methode *on* des *errors*-Objekts der Active Record-Klasse *user*. Die Methode liefert ein Array mit Fehlermeldungen, die sich auf das übergebene Attribut beziehen (hier *name*).

validates_numericality_of

Überprüft, ob der Wert des angegebenen Attributs numerisch ist.

```
class User < ActiveRecord::Base
  validates_numericality_of :age
end
```

validates_uniqueness_of

Prüft die Eindeutigkeit von Attributwerten. Z.B. müssen wir sicherstellen, dass Projektnamen immer eindeutig vergeben werden:

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

Project.create(:name => "OnTrack")
project = Project.new(:name => "OnTrack")
assert !project.save
```

Der *validates_uniqueness_of*-Aufruf sorgt dafür, dass sich das zweite OnTrack-Projekt nicht speichern lässt.

4.15 Callbacks

Callback-Handler ermöglichen den Eingriff in den Lifecycle von Active Record-Instanzen. Wir können also bestimmte Dinge tun, bevor oder nachdem ein Objekt seinen Zustand verändern wird bzw. verändert hat. Callback-Handler beginnen mit dem Präfix *after_* oder *before_*, gefolgt vom Namen der eigentlichen Methode. Vor dem Speichern wird also *before_save* und nach dem Speichern *after_save* aufgerufen.

Callback-Handler befinden sich in einer Queue und werden hintereinander vor und nach der eigentlichen Aktion ausgeführt. Tabelle 4.5 liefert einen Überblick über die vorhandenen Handler sowie deren Aufrufreihenfolge.

Before-Handler liefern einen Booleschen Wert. Liefert einer der Handler der Queue den Wert *false*, werden alle nachfolgenden Handler sowie die eigentliche Aktion nicht mehr ausgeführt.

Neben den *before*- und *after*-Callback-Paaren enthält Active Record zwei weitere Callbacks: *after_find* und *after_initialize*. Der Callback-Handler *after_find* wird nach jedem *find*-Aufruf und *after_initialize* nach der Erzeugung einer Active Record-Instanz aufgerufen.

Tabelle 4.5: Active Record-Callbacks

Neues Objekt: o.save	Vorhandenes Objekt: o.save	o.destroy
1. before_validation()	1. before_validation()	1. before_destroy()
2. before_validation_on_create()	2. before_validation_on_update()	2. destroy()
3. after_validation()	3. after_validation()	3. after_destroy()
4. after_validation_on_create()	4. after_validation_on_update()	
5. save()	5. save()	
6. before_save()	6. before_save()	
7. before_create()	7. before_update()	
8. after_create()	8. after_update()	
9. after_save()	9. after_save()	

4.15.1 Überschreiben von Callback-Methoden

Die einfachste Möglichkeit, in den Lifecycle von Active Record-Objekten einzugreifen, ist das Überschreiben der entsprechenden Callback-Methoden. Als Beispiel soll unsere Projektmanagement-Software um eine E-Mail-Funktion erweitert werden: Immer dann, wenn ein Task verändert wurde, soll der Projektleiter eine E-Mail erhalten. Wir implementieren diese Funktion durch Überschreiben der Methode *after_save*. Die Methode wird nach jedem Speichern eines Objekts aufgerufen, egal ob es sich dabei um ein neues oder ein zu aktualisierendes Objekt handelt:

```
class Task < ActiveRecord::Base
  protected
    def after_save
      TaskMailer.send_status_change(self.status)
    end
end
```

4.15.2 Callback-Makros

Eine weitere Möglichkeit zur Nutzung der Callbacks ist die Verwendung von Callback-Makros. Callback-Makros sind Klassenmethoden, die im Deklarationsteil einer Active Record-Klasse aufgerufen werden. Der Name des Makros entspricht dem Namen der gewünschten Callback-Methode.

Im Gegensatz zum Überschreiben von Callback-Methoden hat die Verwendung von Callback-Makros den Vorteil, dass die von einem Callback hinzugefügte Funktionalität über die Vererbungshierarchie hinweg erhalten bleibt. Dazu ein Beispiel:

```
class Person < ActiveRecord::Base
  before_destroy :destroy_documents
  ...
end

class Manager < Person
  before_destroy :destroy_secret_documents
  ...
end
```

Das in der Klasse *Person* installierte Makro sorgt dafür, dass vor dem Löschen einer Person alle mit der Person assoziierten Dokumente gelöscht werden. Für Manager muss zusätzlich sichergestellt sein, dass neben den „normalen“ Dokumenten eines Managers auch alle von ihm erstellten Geheimdokumente gelöscht werden.

Würde man beide *before_destroy*-Callbacks nicht wie im obigen Codeausschnitt in Form von Makros, sondern durch ein Überschreiben der Methode *before_destroy* installieren, dann wäre die in *Person* hinzugefügte Callback-Funktionalität in der Klasse *Manager* nicht mehr verfügbar. Hingegen sorgt das Hinzufügen der Callback-Funktionalität mit Hilfe eines Makros dafür, dass für Manager sowohl der Handler *destroy_documents* als auch der Handler *destroy_secret_documents* aufgerufen wird. Es werden also nicht nur die geheimen Dokumente, sondern auch alle anderen Dokumente eines Managers gelöscht.

Makros fügen damit wirklich neue Funktionalität hinzu, während das Überschreiben von Handlern eher ein Ersetzen der bestehenden Funktionalität ist.

Es gibt drei unterschiedliche Nutzungsarten von Callback-Makros: Callback-Objekte, Methoden-Referenzen und Inline-Methoden.

Callback-Objekte

Callback-Objekte implementieren genau die Callback-Methoden, die sonst in der Active Record-Klasse überschrieben werden. Durch die Verwendung von Callback-Objekten erreichen wir eine bessere Trennung von Kernfunktionalität der Modellklasse und Sekundär-Funktionalität, wie das Versenden von E-Mail (*separation of concerns*). Callback-Objekte werden durch den Aufruf des Callback-Makros mit dem Objekt als Parameter installiert:

```
class Task < ActiveRecord::Base
  after_save TaskMailer.new
end
```

In der Callback-Klasse selbst werden alle Methoden implementiert, die in der Active Record-Klasse für die Installation des Objekts aufgerufen wurden. Beispiel: In der *Task*-Klasse wurde eine *TaskMailer*-Instanz durch Aufruf der Methode *after_save* installiert. Dementsprechend muss die Methode *after_save* in der Klasse *TaskMailer* implementiert werden:

```
class TaskMailer < ActionMailer::Base
  def after_save record
    TaskMailer.send_status_change record.status
  end
  ...
end
```

Die Methoden von Callback-Objekten erhalten als Parameter die Instanz übergeben, auf der die eigentliche Active Record-Methode aufgerufen wurde. Bezogen auf das obige Beispiel bedeutet dies, dass der Parameter *record* genau die *Task*-Instanz ist, auf der zuvor die Methode *save* aufgerufen wurde.

Methoden-Referenzen

Methoden-Referenzen werden einem Callback-Makro als Symbol übergeben:

```
class Task < ActiveRecord::Base
  before_save :before_save_handler

  def before_save_handler
    ...
  end
end
```

Inline-Methoden

Bei der Verwendung von Inline-Methoden wird dem Callback-Makro ein String übergeben, der bei Ausführung des Handlers evaluiert wird:

```
class Iteration < ActiveRecord::Base
  before_destroy 'Task.delete_all "iteration_id = #{id}"'
end
```

4.15.3 Observer

Die Verwendung von Callback-Objekten war bereits ein erster Schritt, um Kernfunktionalität von Sekundär-Funktionalität zu trennen. Ein noch weiter gehender Schritt in diese Richtung ist die Verwendung von *Observern*.

Observer-Klassen reagieren auf Lifecycle-Ereignisse außerhalb der eigentlichen ActiveRecord-Klasse. Observer-Klassen werden im Verzeichnis *app/models* des jeweiligen Rails-Projekts gespeichert und erben von *ActiveRecord::Observer*. Observer und Modelle werden über Namenskonventionen miteinander verbunden, indem eine Observer-Klasse den Namen der Modellklasse, gefolgt vom Wort *Observer*, trägt. Für die Bearbeitung von Callback-Ereignissen implementiert ein Observer analog zu einer ActiveRecord-Klasse entsprechende Callback-Methoden:

```
class TaskObserver < ActiveRecord::Observer
  def after_save(model)
  end
end
```

Soll ein Observer andere als die über die Namenskonvention ausgedrückten ActiveRecord-Instanzen beobachten, funktioniert das über die Methode *observe*:

```
class MailObserver < ActiveRecord::Observer
  observe Task

  def after_save(model)
    TaskMailer.send_status_change record.status
  end
end
```

4.16 Konkurrierende Zugriffe und Locking

Konkurrierende Zugriffe sind eines der klassischen Probleme von Datenbank-basierten Mehrbenutzersystemen. Nehmen wir an, zwei Benutzer öffnen gleichzeitig denselben Task im Editier-Modus, bearbeiten ihn und speichern den Task anschließend zu unterschiedlichen Zeitpunkten. Wir müssen uns überlegen, wie unsere Applikation mit derartigen Konkurrenzsituationen umgehen soll.

Das Active Record-Standardverhalten bei konkurrierenden Zugriffen besteht darin, dass der zuletzt ausgeführte Schreibaufruf gewinnt, d.h. die vom ersten Aufruf durchgeführten Änderungen werden überschrieben. Der folgende Codeausschnitt demonstriert dieses Verhalten:

```
t1 = Task.find(:first)
t2 = Task.find(:first)

t1.update_attribute("name", "Projektstruktur")
t2.update_attribute("name", "Infrastruktur")
assert_equal "Infrastruktur", Task.find(:first).name
```

Obwohl die Aktualisierung von *t2* auf einer veralteten Version erfolgt, gewinnt *t2*, d.h. die Änderungen von *t1* werden überschrieben. Dieses Verhalten ist nicht immer gewünscht und kann durch die Verwendung von optimistischem Locking unterbunden werden.

4.16.1 Optimistisches Locking

Optimistisches Locking erlaubt mehreren Parteien das gleichzeitige Bearbeiten ein und derselben Modellinstanz.¹¹ Allerdings gewinnt beim optimistischen Locking der erste Schreiber, d.h. nachdem der erste Bearbeiter gespeichert hat, schlagen alle folgenden save-Aufrufe fehl und resultieren in einer Exception.

Optimistisches Locking wird durch die Erweiterung der betroffenen Tabellen um das zusätzliche Feld *lock_version* eingeschaltet:

```
class AddLockVersionToTasks < ActiveRecord::Migration
  def self.up
    add_column :tasks, :lock_version, :integer, :null => false,
              :default => 0
  end

  def self.down
    remove_column :tasks, :lock_version
  end
end
```

¹¹Im Gegensatz dazu würde pessimistisches Locking nur der ersten Partei erlauben, den Datensatz zu editieren, indem er gelockt wird. Alle weiteren Editieranfragen würden so lange abgewiesen, bis der erste Bearbeiter den Satz gespeichert und wieder freigegeben hat.

Der Default-Wert des Felds sollte 0 sein. Führen wir jetzt den Testcode aus dem vorigen Beispiel erneut aus, erhalten wir statt eines positiven Testreports die folgende Exception:

```
1) Error:
test_default_locking(LockingTest):
ActiveRecord::StaleObjectError: Attempted to update a
                                stale object
```

Was passiert hier? Active Record verwendet das neue Feld *lock.version* als interne Versionsnummer eines Datensatzes, die bei jedem Speichern automatisch um 1 inkrementiert wird. Vor dem Speichern prüft Active Record allerdings, ob die Versionsnummer des Datensatzes mit der zugehörigen Modellinstanz übereinstimmt. Ist die Nummer in der Datenbank größer, hat ein anderer Bearbeiter das Objekt vorher gespeichert, und der aktuelle Speicherversuch endet mit einer Exception.

Kapitel 5

Action Controller

Rails Action Controller repräsentieren das *C* im zugrunde liegenden MVC-Pattern. Controller steuern den Kontrollfluss einer Rails-Anwendung. Sie nehmen HTTP-Requests entgegen, bearbeiten diese und senden anschließend einen HTML-View an den Client zurück.

Action Controller sind Teil von *Action Pack*, einem der drei Sub-Frameworks von Rails. Der zweite Bestandteil von Action Pack ist *Action View*. Action View implementiert die View-Funktionalität von Rails und wird in Kapitel 6 ausführlich beschrieben.

5.1 Controller-Grundlagen

Rails-Controller sind normale Ruby-Klassen, die von der Klasse *ApplicationController* erben. Diese Klasse wird vom Rails-Anwendungsrahmen bereitgestellt und dient als Basisklasse für alle anwendungsspezifischen Controller. Sie erbt wiederum von der Framework-Klasse *ActionController::Base*:

```
class ProjectsController < ApplicationController
end

class ApplicationController < ActionController::Base
end
```

Die initiale Version einer Controllerklasse wird durch den Rails-Generator erzeugt:

```
$ ruby script/generate controller projects
  exists  app/controllers/
  exists  app/helpers/
  create  app/views/projects
  exists  test/functional/
  create  app/controllers/projects_controller.rb
  create  test/functional/projects_controller_test.rb
  create  app/helpers/projects_helper.rb
```

Der Generator erzeugt neben der Controllerklasse eine zugehörige Testklasse sowie ein Helper-Modul. Die Verwendung des Generators ist nicht zwingend, d.h. Sie können die Controllerklasse genau wie jede andere Klasse von Hand programmieren.

Im Beispiel geben wir den Plural des Namens an, also *projects* und nicht *project*. Wir folgen damit der REST-Namenskonvention von Rails, auf dem das REST-Routing beruht (vgl. Abschnitt 7.10). Sie können auch den Singular des Namens verwenden, aber wir empfehlen Ihnen das nicht.

5.1.1 Actions

Für die Bearbeitung von HTTP-Requests implementiert ein Controller so genannte *Actions*. Eine Action ist eine öffentliche Methode der Controllerklasse, die einen bestimmten HTTP-Request entgegen nimmt.

```
class ProjectsController < ActionController::Base
  def hello
  end
end
```

Das Codebeispiel definiert die Action *show*, wodurch der *ProjectsController* jetzt in der Lage ist, den HTTP-Request an *show* zu bearbeiten.

Der Aufruf von Actions erfolgt über Reflection und erfordert keine Konfiguration. Dazu parst Rails die empfangene URL, extrahiert Controller- und Action-Name daraus und ruft die so bestimmte Methode dynamisch auf (siehe Abbildung 5.1).

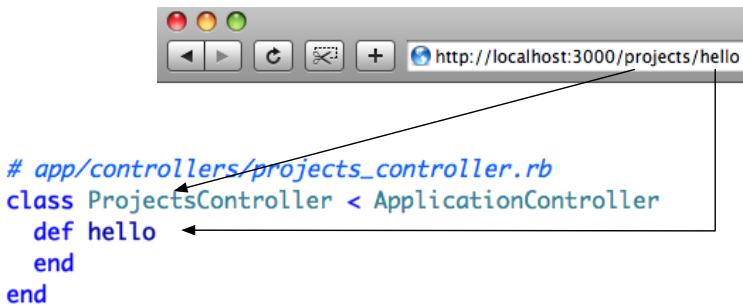


Abbildung 5.1: Rails Controller- und Action-Name aus der URL

Enthält die URL neben Controller- und Actionnamen einen weiteren (namenlosen) Parameter, dann wird dieser Parameter unter dem Bezeichner *id* an die zu aktivierende Action übergeben. Beispielsweise können Sie der *show*-Action des *ProjectsControllers* die ID des anzuzeigenden Projektes durch Eingabe der URL *http://localhost:3000/projects/show/1* mitteilen.

Auf den ersten Blick mag es Ihnen an dieser Stelle vielleicht ein wenig verwirrend erscheinen, dass die Zahl 1 als Request-Parameter mit dem Bezeichner *id* an den Controller übergeben wird, obwohl das Wort *id* an keiner Stelle in der URL auf-

taucht. Verantwortlich für dieses Verhalten ist der dem Controller-Framework zugrunde liegende Routing-Mechanismus, der die einzelnen Komponenten einer URL auf entsprechende Parameter im HTTP-Request abbildet. In einem generierten Rails-Anwendungsrahmen ist das Routing so konfiguriert, dass der dritte Parameter einer URL, sofern vorhanden, immer unter dem Bezeichner *id* in den Request-Parametern auftaucht. Detaillierte Informationen über Routing und dessen Konfiguration finden Sie in Abschnitt 5.12.

Der Controller kann die ID jetzt über den Schlüssel *id* aus der *params*-Hash extrahieren und für das Laden des gewünschten Projekts verwenden:

```
class ProjectsController < ActionController::Base
  def show
    @project = Project.find(params[:id])
  end
end
```

Alle öffentlichen Methoden einer Controllerklasse sind Actions und können demzufolge von außen über einen HTTP-Request aufgerufen werden. Es gibt zwei Möglichkeiten, dieses Verhalten abzustellen. Entweder Sie deklarieren die Methode als *private* bzw. *protected*,¹ oder Sie markieren die Methode mittels *hide_action* als verborgen:

```
class ProjectsController < ActionController::Base
  hide_action :validate_project

  def validate_project
    ...
  end
end
```

Grundsätzlich ist die Verwendung von *private* oder *protected* vorzuziehen. Trotzdem gibt es Gründe für die Deklaration öffentlicher Methoden, ohne dass diese Methoden gleich zu Actions werden sollen. Ein solcher Grund ist die Testbarkeit, z.B. dann, wenn wir einen Unit Test für die Methode *validate_project* schreiben wollen.²

5.1.2 Responses

Im Anschluss an die Ausführung einer Action liefert ein Controller eine Antwort an den Browser. Im Standardfall handelt es sich hierbei um ein RHTML-Template (vgl. Kapitel 6), d.h. eine HTML-Seite mit eingebettetem Ruby-Code, die vor der endgültigen Auslieferung an den Browser noch vom ERb³-Prozessor bearbeitet wird. Alternativ zu einem RHTML-Template kann eine Action auch einen einfachen Text liefern bzw. ein anderes Datenformat als HTML (Word, PDF etc.) an den Browser senden.

¹ Vgl. Kasten *Ruby: Sichtbarkeit von Methoden* in Abschnitt 3.15.

² Wir sind uns bewusst, dass die Veröffentlichung von Methoden zu Gunsten besserer Testbarkeit kontrovers diskutiert wird. Wir haben für uns entschieden, dass die Aufweichung bestimmter OO-Prinzipien in Ordnung ist, wenn wir unseren Code dadurch besser testen können.

³ Embedded Ruby

Die Methode render

Wir beginnen mit dem Standardfall, dem Ausliefern eines RHTML-Views. Für diesen einfachsten Fall müssen Sie gar nichts tun. Rails liefert standardmäßig den View, dessen Name dem Namen der gerade ausgeführten Action entspricht. Die folgende *show*-Action enthält keine explizite Angabe eines Views, woraufhin der Controller im Anschluss an deren Ausführung den View *show.html.erb* ausliefert:

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

Der *ProjectsController* erwartet, dass der View *show.html.erb* im Verzeichnis *on-track/app/views/projects* liegt (siehe Abbildung 5.2).

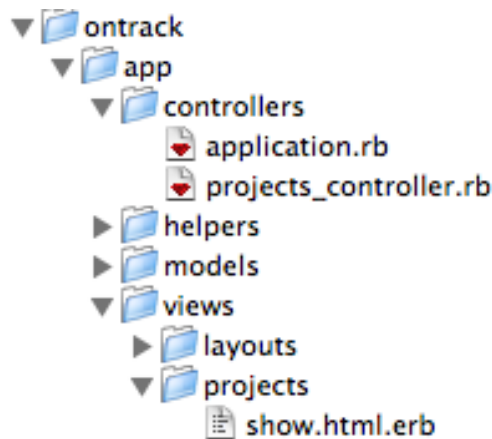


Abbildung 5.2: Der *ProjectsController* und seine Views im Verzeichnis *app/views/projects*

Das View-Basisverzeichnis *app/views* ist von Rails vorgegeben, kann aber durch Aufruf der Methode *ActionController::Base.template_root* explizit gesetzt werden.

In der Praxis kommt es häufig vor, dass der Name des darzustellenden Views vom Namen der aktuellen Action abweicht. Für diesen Fall besitzt jeder Controller die Methode *render*, deren Parameter *:action* den darzustellenden View vorgibt. Im folgenden Beispiel liefert die *show*-Action den View *special_show.html.erb*, der im Verzeichnis *app/views/projects* liegen muss:

```
def show
  @project = Project.find(params[:id])
  render :action => "special_show"
end
```

Wenn sich der View in einem anderen als dem View-Verzeichnis des Controllers befindet, können Sie durch Verwendung des Symbols *:template* den Namen des Views zusammen mit dem zugehörigen Basisverzeichnis (relativ zum Verzeichnis *app/views*) vorgeben. Der nachfolgende *render*-Aufruf erwartet den View *special_show.html.erb* im Verzeichnis *app/views/common/*.

```
render :template => "common/special_show"
```

Alternativ können Sie über *:file* auch einen absoluten Pfad angeben:

```
render :file => "/path/to/file/show.html.erb"
```

Eine schnelle Möglichkeit zum Testen der Funktionstüchtigkeit einer Action ist die Verwendung des *:text*-Symbols. Der folgende *render*-Aufruf sendet den Text *Hello, Rails* an den Client:

```
render :text => "Hello, Rails"
```

Hin und wieder kommt es vor, dass eine Action kein Ergebnis liefert, d.h. weder einen View an den Client sendet, noch ein Redirect auf eine andere Action ausführt. Ein typisches Beispiel für diese Art von Actions sind Ajax-Aufrufe (siehe z.B. Abschnitt 10.6). Für diese Art von Anwendungsfällen besitzt *render* den Parameter *:nothing*:

```
render :nothing => true
```

Eine weitere Möglichkeit zur Benutzung von *render* im Zusammenhang mit RJS-Templates ist die Option *:update*, die in Abschnitt 10.8.2 ausführlich beschrieben wird.

Weitere Parameter von render

Jeder *render*-Aufruf akzeptiert zusätzlich die beiden optionalen Parameter *:status* und *:layout*. Mit Hilfe des *:status*-Parameters können Sie den Response-Code im HTTP-Response explizit setzen.

Der Parameter *:layout* bestimmt das Layout, mit dem die Response der Action angezeigt wird. Wir beschreiben das Layout von HTTP-Responses und damit auch diesen Parameter gesondert in Abschnitt 5.13.

Wann endet eine Action?

Die Ausführung einer Action ist nach dem Aufruf von *render* nur dann wirklich beendet, wenn danach keine weiteren Anweisungen mehr folgen. Dazu ein Beispiel:

```
def hello
  render :action => "hello"
  STDOUT.puts "Hier geht es noch weiter"
end
```

Ein Blick in die Log-Ausgaben bestätigt uns, dass Rails zum einen den angegebenen View anzeigt, zum anderen aber auch den Text „Hier geht es noch weiter“ in die Server-Konsole schreibt:

```
Hier geht es noch weiter
127.0.0.1 - - [13/Jul/2006:11:44:51 CEST] "GET /projects/
hello HTTP/1.1" 200 0
- -> /projects/hello
```

Die Tatsache, dass eine Methode nicht nach Ausführung des *render*-Aufrufs beendet ist, wird insbesondere dann problematisch, wenn eine Action mehr als einen *render*-Aufruf enthält:

```
def double_render
  begin
    render :action => "hello" if logged_in?
    raise "ein Fehler"
  rescue
    render :text => "Bitte melden Sie sich an"
  end
end
```

Der Code resultiert in einer *DoubleRenderError*-Exception, da die Exception in jedem Fall geworfen wird, also auch dann, wenn der Benutzer bereits am System angemeldet ist. Der Code des obigen Beispiels ist also fehlerhaft und in dieser Form nicht brauchbar. Als Konsequenz daraus sollten Actions explizit so entwickelt werden, dass der Kontrollfluss immer nur genau einen *render*- bzw. *redirect_to*-Aufruf (vgl. Abschnitt 5.3) durchlaufen kann.

5.2 Datenaustausch

Web-Anwendungen bestehen nicht nur aus statischen HTML-Seiten, sondern aus einer Vielzahl dynamischer Seiten, deren Inhalte sich in Abhängigkeit von Datenbankinhalten oder Benutzerinteraktion ständig ändern. Ein View benötigt also neben statischen HTML-Informationen dynamische Daten, die ihm vom Controller zur Verfügung gestellt werden.

5.2.1 Vom Controller zum View

Der Austausch von Daten zwischen Controllern und Views erfolgt in Rails über die Nutzung von Instanzvariablen. Dies bedeutet, dass jedem View sämtliche Instanzvariablen des zugehörigen Controllers zur Verfügung stehen.

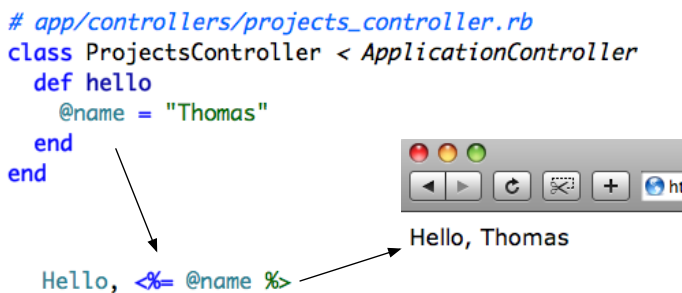


Abbildung 5.3: Der *ProjectsController* versorgt den View mit Daten

Abbildung 5.3 veranschaulicht das Prinzip des Datenaustauschs: Der *Project-Controller* instanziiert die Instanzvariable *@name* in seiner *hello*-Action und liefert anschließend den zugehörigen *hello*-View. Der *hello*-View greift auf die Variable *@name* über ganz normalen Ruby-Code zu. RHTML-Seiten werden vor ihrer Auslieferung an den Client vom ERb-Prozessor bearbeitet, der den enthaltenen Ruby-Code ausführt und als Ergebnis eine für den Client bestimmte HTML-Seite liefert.

5.2.2 Vom View zum Controller

Genau wie ein Controller den View mit Daten versorgt, müssen die Daten eines Views an den Controller übermittelt werden. Web-Anwendungen bestehen typischerweise aus vielen Formularen, in denen der Benutzer Daten eingeben und an den Server übertragen kann. Eine andere Variante der Datenübertragung aus dem Browser zum Controller ist die Verwendung von URL-Parametern. Beide Varianten werden wir im Folgenden als Request-Parameter bezeichnen.

Der Zugriff auf die Daten eines vorausgehenden HTTP-Requests erfolgt über die Getter-Methode *params*, die eine Hash mit den Parametern des Requests liefert:

```
class ProjectsController < ApplicationController
  def update
    projectname = params[:projectname]
  end
  ...
end
```

Abbildung 5.4 zeigt, wie es geht: Der View deklariert zwei Eingabefelder mit den Namen *projectname* und *description*.⁴ Die Formulardaten werden an die Controller Action *update* übergeben. Innerhalb der Action stehen die Daten durch Aufruf des Getters *params* zur Verfügung. Die Methode liefert eine Hash, deren Schlüssel den Namen der im View verwendeten HTML-Felder entsprechen. Wird das in der Abbildung dargestellte Formular an den Server übertragen, dann erhält die *update*-Action folgende *params*-Hash:

```
params = {"projectname" => "Taskman",
          "description" => "Taskmanagement System"}
```

Alternativ können die Request-Parameter auch in der URL übergeben werden:

```
http://localhost:3000/projects/update?projectname="Taskman"&
description="Taskmanagement System"
```

⁴Das Formular in Abbildung 5.4 verwendet für die Darstellung von Eingabefeldern und des Submit-Buttons Standard-HTML-Tags. Dies ist in Rails nicht üblich, da Rails für diesen Zweck so genannte Formular-Helfer (siehe Kapitel 6: *Action View*) zur Verfügung stellt. Wir haben uns in diesem Beispiel dennoch für Standard-HTML-Tags entschieden, da so die Funktionsweise des Datenaustauschs zwischen View und Controller verständlicher wird.

Die von Rails erzeugte Hash *params* ist in beiden Fällen identisch.

```
<form action="/projects/update" method="post">
  Name: <input name="projectname" type="text" /><br/>
  Desc: <textarea name="description" /><br/>
  <input type="submit" value="Speichern" name="commit" />
</form>

class ProjectsController < ApplicationController
  def update
    projectname = params[:projectname]
    description = params[:description]
    render :text => projectname
  end
end
```



Abbildung 5.4: Der *ProjectsController* holt sich seine Daten über die Hash *params*

Mehrdimensionale Parameter und Massenzuweisung

Request-Parameter können mehrdimensionale Hashes sein, was insbesondere für die direkte Aktualisierung von Modellen im Controller interessant ist.

```
<form ...>
  <input type="text" name="project[name]" />
  <textarea name="project[description]" />
  ...
</form>
```

Der HTML-Code im obigen Beispiel erzeugt und übergibt folgende *params*-Hash an den Controller:

```
params = {"project" =>
  {"name" => "Taskman",
    "description" => "Taskmanagent System",
    ... }}

```

Die *params*-Hash enthält unter dem Schlüssel *project* wiederum ein Hash, die alle Parameter zum Projekt definiert. In der zugehörigen Controller-Action können Sie diese zweidimensionale Hash für die direkte Aktualisierung des Modells benutzen. In diesem Fall sprechen wir auch von Massenzuweisung (engl. Mass Assignment):

```
def update
  project = Project.find(params[:id])
  project.update_attributes(params[:project])
end
```

Diese Art der Zuweisung ist einfach und elegant und stellt daher den Standard in Rails-Anwendungen dar. Insofern erzeugen viele der Formular-Helfer aus Abschnitt 6.3.1 HTML-Code für diese Art der Parameterübertragung.

Massenzuweisung schützen

Die Massenzuweisung birgt allerdings ein Sicherheitsrisiko. Es sei angenommen, dass das Modell *Person* z.B. ein Feld *admin* enthält, welches einen Anwender als Administrator oder als normalen Benutzer der Anwendung kennzeichnet. Das Feld wird per Default auf 0 gesetzt.

Ein Angreifer kann sich diesen Umstand zunutze machen, indem er das Registrierungsformular an die Anwendung schickt und das Attribut *admin* auf den Wert 1 (true) setzt und somit zum Administrator wird. Er bedient sich dazu eines Programms wie z.B. *curl*⁵, mit dem er den folgenden POST-Request ausführt:

```
$ curl -d "person[name]=MrEvil&person[admin]=1"
http://meinedomain.de/person/sign_up
```

Das beschriebene Szenario lässt sich vermeiden, indem „schützenswerte“ Attribute in der Modellklasse entsprechend deklariert werden:

```
class Person < ActiveRecord::Base
  attr_protected :admin
end
```

Geschützte Attribute werden beim Erzeugen oder Aktualisieren nicht berücksichtigt und müssen entsprechend explizit gesetzt werden, sofern dies erwünscht ist:

```
# person.admin wird nicht gesetzt!
person = Person.create(params[:person])

# muss also explizit gesetzt werden, sofern gewünscht
person.admin = params[:person][:admin]
```

Alternativ zum expliziten Ausschluss sensibler Attribute können die aktualisierbaren Attribute explizit angegeben werden. Alle anderen Modellattribute sind dann von der Zuweisung ausgeschlossen:

```
class Person < ActiveRecord::Base
  attr_accessible :name, :fon, :fax
end
```

5.2.3 Aus der Action in den View und zurück

Selbstverständlich können Sie den Datenaustausch zwischen Controller und View in beide Richtungen kombinieren. Beispielsweise dienen Formulare nicht nur der Erfassung neuer Daten, sondern zeigen i.d.R. bereits vorhandene Daten an. Dazu müssen Sie für das Attribut *value* Ihrer Eingabefelder einfach die im Controller erzeugten Instanzvariablen verwenden:

```
<input type="text" name="projectname" value="<%= @name %>" />
```

Noch einfacher wird es, wenn Sie den Rails-Formular-Helper *text_field* verwenden, der das *value*-Attribut automatisch mit dem zugehörigen Attributwert eines vom Controller instanziierten Modells füllt:

⁵ <http://curl.haxx.se>

```
<%= text_field :project, :name %>
```

In diesem View-Fragment wird das Eingabefeld mit dem Wert des Attributs *name* vorbelegt. Das Attribut bezieht sich dabei auf die Instanzvariablen *@project* des zugehörigen Controllers. Keine Angst: Wem diese Beschreibung etwas zu schnell war, der findet eine ausführliche Beschreibung dieses und aller weiteren Formular-Helper im Kapitel 6: *Action View*.

5.3 Redirects

Ein Redirect⁶ bezeichnet die Weiterleitung eines HTTP-Requests an eine andere Action. Die Weiterleitung findet keinesfalls nur serverseitig statt, sondern nimmt einen Umweg über den Browser, indem die Action dem Browser eine Redirect-Response sendet. Der Browser handhabt Redirects intern, d.h. er erzeugt automatisch einen neuen HTTP-Request, ohne dass der Anwender dies direkt bemerkt, außer natürlich an der Anzeige einer neuen URL, auf die weitergeleitet wurde. Warum das Ganze?

Typischerweise liefert eine Action im Anschluss an ihre Ausführung einen View (vgl. Abschnitt 5.1.2). Ausnahmen bilden Actions, die neue Modellinstanzen erzeugen und speichern oder anderweitig schreibend auf die Datenbank zugreifen. Dazu ein Beispiel: Die Action *add_iteration* fügt einem Projekt eine neue Iteration hinzu. Anschließend soll das Projekt mit einer Übersichtsliste aller Iterationen des Projekts angezeigt werden. Eine mögliche Lösungsvariante wäre die Anzeige des *show*-Views am Ende der Ausführung von *add_iteration*:

```
class ProjectsController < ActionController::Base

  def add_iteration
    @project = Project.find(params[:id])
    @project.iterations.create(params[:iteration])
    render :action => "show"
  end

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

Alles wunderbar: Der Controller aktualisiert das Projekt und versorgt den *show*-View über die Bereitstellung der Instanzvariable *@project* mit aktuellen Daten. Im Browser bleibt jedoch weiterhin die URL des ursprünglichen *update*-Requests stehen (*http://localhost:3000/projects/add_iteration*), was dazu führt, dass die Action ein weiteres Mal aufgerufen wird, wenn der Benutzer die Seite im Browser neu lädt. Dies wiederum führt dazu, dass eine weitere Iteration erzeugt und dem Projekt hinzugefügt wird. Mit jedem Neuladen der Seite wird die Liste der Iterationen also um einen Eintrag länger.

⁶ auf Deutsch „Weiterleitung“

Deshalb gilt: Eine Action, die ein Modell erzeugt oder ändert, sollte nach dem Speichern des Modells nicht zusätzlich für dessen Anzeige zuständig sein. Diese Aufgabe sollte von einer Action *show* übernommen werden, auf die aus der *add_iteration*-Action weitergeleitet wird.

Zur Durchführung von Redirects steht die Methode *redirect_to* zur Verfügung. Die Methode erwartet als Parameter den Namen der Action, auf die weitergeleitet wird:

```
def add_iteration
  @project = Project.find(params[:id])
  @project.iterations.create(params[:iteration])
  redirect_to :action => "show", :id => @project
end
```

Neben dem Action-Namen akzeptiert *redirect_to* beliebige weitere Parameter, auf die die Ziel-Action über die *params*-Hash Zugriff hat. Beispielsweise übergibt der *redirect_to*-Aufruf des obigen Beispiels neben dem Action-Namen die ID des darzustellenden Projekts. Die *show*-Action verwendet diese ID zum Laden des Projekts und liefert anschließend den *show*-View (implizit):

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

Wenn ein Benutzer die Seite im Browser jetzt neu lädt, wird das Projekt nicht nochmals gespeichert, sondern nur neu geladen und dargestellt.

Befindet sich die Ziel-Action des Redirects nicht in derselben Controllerklasse, so muss die Controllerklasse explizit durch Verwendung des Parameters *:controller* angegeben werden:

```
redirect_to :action => "show", :controller => "projects"
```

Sie können für *redirect_to* eine Named Route (vgl. 5.12.4) verwenden. Wir empfehlen Ihnen diese Art der Parameterangabe:

```
redirect_to project_path(@project)
```

Eine letzte Möglichkeit ist die Verwendung mit einer absoluten URL als Parameter:

```
redirect_to "http://projects/show"
```

Für Redirects gilt im Übrigen das Gleiche wie für das Anzeigen von Views: Die Action ist nach Ausführung des *redirect_to*-Aufrufs nicht beendet, was bei falscher Benutzung in *DoubleRenderError*-Exceptions resultiert.

5.4 GET vs. POST

Das HTTP-Protokoll unterscheidet u.a. zwischen GET- und POST-Requests (vgl. Kapitel 7). GET-Requests dienen der Datenübertragung vom Server zum Client, z.B. eine Anforderung einer Liste aller im System erfassten Projekte. Diese Requests müssen idempotent sein, d.h. sie dürfen den Zustand des Systems (z.B. die Daten

in der Datenbank) nicht verändern, egal wie häufig sie (hintereinander) aufgerufen werden. POST-Requests sind Anforderungen für Zustandsänderungen auf dem Server, wie z.B. die Erzeugung neuer oder das Verändern vorhandener Daten. HTTP-Links und damit auch der Active View Helper *link_to* erzeugen per Default GET-Requests.

Spider oder Suchmaschinen durchsuchen öffentliche Web-Sites und folgen den enthaltenen Links. Das Problem dabei: Viele Web-Anwendungen beachten nicht die eingangs beschriebene Regel und lösen Zustandsänderungen auf dem Server über GET-Requests aus. Konkret bedeutet dies, dass ein Spider durch die simple Verfolgung von Links in der Lage ist, den Zustand der Anwendung zu ändern. POST-Requests kennen dieses Problem nicht, da sie von Spidern und Suchmaschinen nicht automatisch abgesetzt werden können.

Im Kontext der Rails-Entwicklung lässt sich die Regel „GET für Anfragen und POST für Zustandsänderungen“ wie folgt einhalten. Für die Views gilt:

- **Links für Datenanfragen.** Anfragen, die ausschließlich Daten vom Server zum Client übertragen, werden durch Links (*link_to*) abgesetzt. Die Angabe des Parameters *:method => :delete* erzeugt hingegen eine Form, die per POST übertragen wird (vgl. Abschnitt 7.8.3).
- **Formulare und Buttons.** Anfragen, die den Zustand des Servers ändern, werden über Formulare (z.B. *form_for*) oder Buttons (*button_to*) an den Server gesendet.
- **Bestätigungsseiten.** Wenn Sie dennoch Links für die Anfrage nach Zustandsänderungen verwenden wollen, empfiehlt sich das Zwischenschalten einer Bestätigungsseite. Beachten Sie bitte, dass der *:confirm*-Parameter der *link_to*-Methode eine Bestätigungsseite nicht ersetzen kann. Spider können diesen Link trotzdem verfolgen.

Im Controller können Sie auf zwei Arten sicherstellen, dass Ihre Action nur von der gewünschten HTTP-Methode aufgerufen wird.

HTTP-Methode des Requests prüfen

Sie können die HTTP-Methode und damit einen POST-Request wie folgt ermitteln und entsprechend reagieren:

```
def create
  render(:file => "#{RAILS_ROOT}/public/404.html",
        :status => "404") and return unless request.post?
  ...
end
```

Auf einen Ajax-Request können Sie mithilfe der Methode *xhr?* prüfen:

```
def add_iteration
  ... and return unless request.xhr? # xml_http_request?
  ...
end
```

Einsatz des Filters `verify`

Da Sie in der Regel mehrere Actions absichern müssen, empfiehlt sich der Einsatz eines Filters (vgl. Abschnitt 5.10), den Rails Ihnen über die Methode `verify` zur Verfügung stellt:

```
class Controller ProjectsController < ApplicationController
  verify :method => :post, :only => [ :create, :update, :destroy ],
        :redirect_to => { :action => :list }
  ...
end
```

Im Beispiel prüft die Methode, ob der Request für die über `:only` definierten Actions vom Typ `:method` (hier POST) ist. Wenn nicht, wird ein Redirect auf die Action `list` durchgeführt, statt die eigentliche Action aufzurufen. Wie Sie hier reagieren, bleibt Ihnen überlassen. Die Methode `verify` kann auch für andere Prüfungen genutzt werden, z.B. ob ein Wert in der Session enthalten ist.

5.5 Unterschiedliche Response-Formate: `respond.to`

Ein HTTP-Client kann im Header-Feld `accept` sein bevorzugtes Response-Format spezifizieren. Das Feld enthält eine Liste von MIME-Typen, deren Reihenfolge die Priorität der gewünschten Antwortformate vorgibt. Ein Client, der im `Accept`-Feld den Wert

```
Accept: */*
```

vorgibt, macht keinerlei Angaben über gewünschte Response-Formate und deren Prioritäten. Spezifiziert ein Client im `Accept`-Feld hingegen folgende Liste:

```
Accept: text/javascript, text/html, application/xml, text/xml */*
```

dann teilt er dem Server mit, dass er sich in erster Linie für JavaScript interessiert, und erst wenn dies nicht zur Verfügung steht, auch mit HTML einverstanden ist.

Mit `respond.to` steht eine einfache Methode zur Auswertung des vom Client gewünschten Response-Formats zur Verfügung. Die Methode eignet sich zur Unterstützung unterschiedlicher Client-Typen, z.B. von „normalen“ Browsern, Ajax-fähigen Browsern oder API-Clients.

Wir demonstrieren die Verwendung von `respond.to` am Beispiel der `create`-Action des `ProjectsControllers`. Zunächst die ursprüngliche Version:

```
class ProjectsController < ApplicationController
  def create
    @project = Project.create(params[:project])
    redirect_to :action => "list"
  end
  ...
end
```

Die Methode *create* wird so erweitert, dass sie außer von normalen Web-Browsern auch von Ajax-fähigen Browsern und API-Clients verwendet werden kann:

```
class ProjectsController < ApplicationController
  def create
    @project = Project.create(params[:project])
    respond_to do |type|
      type.html { redirect_to :action => "list" }
      type.js
      type.xml do
        render(:xml => @project.to_xml, :status => "201 Created")
      end
    end
  end
  ...
end
```

Je nach Inhalt des Accept-Felds im HTTP-Header liefert *create* nun das vom Client bevorzugte Antwortformat: HTML, JavaScript oder XML. Im Falle von JavaScript (type.js) wird auf die Angabe einer expliziten Response verzichtet, sodass standardmäßig das RJS-Template *create.rjs* ausgeliefert wird (siehe auch 10.8).

Sie können die neue *create*-Action am Beispiel eines API-Clients testen, der die Daten des neu zu erzeugenden Projekts als XML sendet:

```
$ curl -H "Content-Type: application/xml" \
> -H "Accept: application/xml" -i -X POST \
> -d "<project><name>OnTrack</name></project>" \
> http://localhost:3000/projects/create
```

```
HTTP/1.1 201 Created
Date: Mon, 10 Mar 2008 09:38:28 GMT
Set-Cookie: _session_id=BAh7BiIKZmxhc2...
Status: 201 Created
Server: Mongrel 1.0.2
...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <name>OnTrack</name>
  <id type="integer">26</id>
  <iterations-count type="integer">0</iterations-count>
</project>
```

Das Kommandozeilen-Tool *curl* ermöglicht u.a. die Kommunikation mit HTTP-Servern. Über den Content-Type *application/xml* teilen wir dem Controller mit, dass die Daten im XML-Format übergeben werden. Rails übersetzt sie automatisch in eine Params-Hash, sodass der *Project.create*-Aufruf in der ersten Zeile der Action weiterhin funktioniert. Zusätzlich spezifizieren wir über das Header-Feld *Accept*, dass wir im Response XML erwarten. Diese Angabe führt dazu, dass der dritte Zweig des *respond_to*-Blocks ausgeführt wird, sodass eine Antwort entsprechend dem folgenden Format erzeugt wird:

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <name>OnTrack</name>
  <id type="integer">26</id>
  <iterations-count type="integer">0</iterations-count>
</project>
```

Mit *respond_to* können Sie Ihre Anwendung auf die Verwendung durch unterschiedliche Clients vorbereiten bzw. dahingehend erweitern. Die Controllerlogik bleibt für alle Clients die gleiche, und Wiederholungen werden vermieden. Wir werden auf dieses Thema im Kapitel 7 nochmals eingehen.

5.6 Zugriff auf Datensätze einschränken

Bei einem Aufruf einer Action wird häufig die ID einer Modellinstanz übergeben und an die *find*-Methode weitergereicht:

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

Niemand hindert einen Benutzer daran, eine andere ID direkt in der URL anzugeben und so Daten eines Projekts zu setzen, das ihm nicht gehört. Sie schützen die Datensätze eines Benutzers, indem Sie dessen ID in der Bedingung des *find*-Aufrufs angeben. Diese steht typischerweise nach der Anmeldung in der Session zur Verfügung (vgl. Abschnitt 5.8):

```
def show
  @project = Project.find(params[:id],
    :conditions => ["user_id = ?", session[:user_id]])
end
```

Einfacher und eleganter geht der Aufruf über die Assoziationsmethoden (vgl. Abschnitt 4.9). Im Beispiel hat ein Benutzer viele Projekte, und ein Projekt gehört zu einem Benutzer. Die Modelle definieren diese Beziehung wie folgt:

```
class Person < ActiveRecord::Base
  has_many :projects
end

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

Die erzeugte Assoziationsmethode *projects* wird in der Action wie folgt genutzt:

```
def show
  current_user = Person.find(session[:user_id])
  @project = current_user.projects.find(params[:id])
end
```


Rails erzeugt durch diesen Aufruf SQL, das den Benutzer berücksichtigt. Ein Blick in die Log-Ausgaben zeigt den Datenbankaufruf:

```
SELECT * FROM `projects`
  WHERE (`projects`.`id` = 42 AND (projects.person_id = 1))
```

Der Code wird noch weiter vereinfacht, wenn *current_user* als Hilfsmethode definiert und in einen Helper ausgelagert wird:

```
module LoginHelper
  def current_user
    Person.find(session[:user_id])
  end
end
```

Die Methode steht dadurch in allen Views zur Verfügung und kann über den *ApplicationController* ebenso in allen Controllern bereitgestellt werden:⁷

```
class ApplicationController < ActionController::Base
  include LoginHelper
  ...
end

class ProjectsController < ApplicationController
  def show
    @project = current_user.projects.find(params[:id])
  end
end
```

Der Aufruf kann auch über mehrere Assoziationsmethoden gehen. Für die Anzeige eines Tasks sieht der Aufruf z.B. wie folgt aus:

```
class TasksController < ApplicationController
  def show
    @task = current_user.projects.find(params[:project_id])
    .tasks.find(params[:id])
  end
end
```

5.7 Ausnahme fangen mit `rescue_from`

In einer Action laden Sie einen Datensatz in der Regel über seine ID:

```
class ProjectsController < ApplicationController
  def show
    @project = Project.find(params[:id])
  end
  ...
end
```

⁷ Dies geschieht ggf. automatisch über den Aufruf *helper :all* im *ApplicationController* (vgl. Abschnitt 6.2).

Existiert zu der übergebenen ID kein Datensatz, wirft ActiveRecord die Ausnahme *ActiveRecord::RecordNotFound* (vgl. Abschnitt 4.4.2). Der Zugriff über die ID erfolgt in der Regel in mehreren Actions eines Controllers. Um das Fangen einer Ausnahme nicht in jeder Action erneut zu programmieren, bietet Rails die Methode *rescue_from*, die im Controller wie folgt angegeben wird:

```
class ProjectsController < ApplicationController
  rescue_from ActiveRecord::RecordNotFound,
    :with => :record_not_found
  rescue_from Project::NotEditable,
    :with => :not_editable

  def edit
    @project = Project.find(params[:id])
  end
  ...
private
  def record_not_found
    ...
  end
  def not_editable
    ...
  end
end
```

Im Falle der Ausnahme *Project::NotEditable* wird die Methode *not_editable* aufgerufen, in der Sie auf die Ausnahme reagieren. Alternativ geben Sie der Definition von *rescue_from* einen Block mit:

```
class ProjectsController < ApplicationController
  rescue_from Project::NotEditable, :with => Proc.new { |ex|
    ...
  }
  ...
end
```

5.8 Sessions

HTTP ist ein zustandsloses Protokoll, d.h. für jeden Request zum Server öffnet der Client eine neue Verbindung. Typische Web-Anwendungen sind in der Regel zustandsbehaftet. Denken Sie nur an das klassische Warenkorb-Beispiel: Ein Benutzer fügt seinem virtuellen Warenkorb einzelne Artikel hinzu. Technisch wird das Hinzufügen durch aufeinanderfolgende HTTP-Requests realisiert, d.h. zwischen dem Hinzufügen von Artikeln geht der aktuelle Zustand verloren, sofern wir nichts dagegen unternehmen.

Um das Problem zu lösen, wird beim ersten Request eine *ShoppingCart*-Instanz erzeugt und in die Datenbank geschrieben. Bei allen weiteren Requests wird die Instanz aus der Datenbank geladen, der Artikel hinzugefügt und anschließend erneut

gespeichert. Schließt der Benutzer den Einkauf ab, wird die *ShoppingCart*-Instanz für gültig erklärt, z.B. indem ein entsprechendes Flag auf *true* gesetzt wird. Bricht der Benutzer den Einkauf ab, bleibt zwar eine „*ShoppingCart*-Leiche“ in der Datenbank. Diese Leichen werden dann periodisch gelöscht. Damit bei jedem Request der Warenkorb eindeutig einem Benutzer zugeordnet werden kann, wird die ID der *ShoppingCart*-Instanz in der Session gespeichert.

Grundsätzlich können Sie jedes serialisierbare Objekt in die Session schreiben.⁸ Sie könnten daher auch die *ShoppingCart*-Instanz direkt in der Session speichern. Wir raten Ihnen jedoch davon ab. Neben möglichen Synchronisationsproblemen zwischen den Objekten in der Session und in der Datenbank leidet auch die Performanz beim Speichern und Laden der Session-Daten pro Request.

Per Default speichert Rails die Session-Daten in einem Cookie. Das ist aus Sicht der Skalierung der Anwendung optimal und sehr performant (vgl. Abschnitt 11.3 und 11.4). Der Cookie selbst kann nur eine begrenzte Menge von Daten aufnehmen (4 KB), aber für die ID und einige kurze Flash-Meldungen (vgl. Abschnitt 5.9) reicht das vollkommen aus. Informationen zur Definition anderer Session-Speicher finden Sie unter Abschnitt 11.4.

Beachten Sie, dass Rails die Übertragung der Session-ID vom Browser an die Anwendung ausschließlich über Cookies realisiert. Der Browser und eine clientseitige Firewall müssen daher das Speichern von Cookies auf dem Rechner der Kunden erlauben.

Die Session steht in jeder Controller-Action als die vom Getter *session* gelieferte Hash zur Verfügung:

```
def show
  ShoppingCart.find session[:cart_id]
end
```

Für das Beispiel des Warenkorbs machen wir uns das wie folgt zu Nutze. Wir erstellen analog Abschnitt 5.6 eine Hilfsmethode *current_cart*, die in allen Controllern und Views zur Verfügung steht und die aktuelle *ShoppingCart*-Instanz liefert. Sofern noch keine Instanz existiert, wird eine neue Instanz erzeugt und deren ID in die Session geschrieben, andernfalls die vorhandene Instanz über die ID geliefert.

```
module ShoppingCartHelper
  def current_cart
    session[:cart_id] ||= ShoppingCart.create.id
    ShoppingCart.find session[:cart_id]
  end
end
```

Um einen Artikel in den Warenkorb zu legen, verwenden wir die Methode *add_article*, die sich den Warenkorb über *current_cart* besorgt:

```
class ShoppingCartsController < ApplicationController
  def add_article
    article = Article.find(params[:id])
```

⁸ Datenbank- oder Netzwerkverbindungen können Sie deshalb nicht in die Session schreiben.

```
        current_cart.add(article)
    end
end
```

Typischerweise wird auch die Anmeldung eines Benutzers an die Anwendung über das Speichern der ID in der Session realisiert. Nach der erfolgreichen Anmeldung wird die ID in die Session geschrieben. Bei jedem weiteren Request prüft die Anwendung, ob es zur übertragenen Session-ID eine Session mit einer Benutzer-ID gibt. Wenn ja, kann der Benutzer die Action ausführen, wenn nicht, wird er auf die Login-Seite geleitet (vgl. Abschnitt 3.15).

Wenn Ihre Anwendung keine Session-Daten benötigt, sollten Sie auf die unnötige Erzeugung von Sessions verzichten. Nutzen Sie dazu die Klassenmethode *session* im *ApplicationController*:

```
class ApplicationController < ActionController::Base
  session :off
end
```

Um die Erzeugung nur für einzelne Actions zu deaktivieren, nutzen Sie die Optionen *only* und *except*:

```
class NewsController < ApplicationController
  session :off, :only => :index
  # session :off, :except => :create
end
```

Über die Methode *session* können eine Reihe weiterer Einstellungen getätigt werden. Werfen Sie einen Blick in die Online-Dokumentation.

5.8.1 Session-Daten löschen

Es gibt zwei Varianten, Daten aus der Session wieder zu entfernen. Die erste besteht im Zuweisen von *nil* an den entsprechenden Session-Key, unter dem das zu löschende Objekt gespeichert ist:

```
def logout
  session[:cart_id] = nil
end
```

Die zweite Variante ist die Verwendung von *reset_session*, die sämtliche Objekte aus der Session entfernt und eine neue Instanz erzeugt:

```
def logout
  reset_session
end
```

5.9 Der Flash-Speicher

Der Flash-Speicher ist ähnlich wie die Session ein Zwischenspeicher zum Austausch von Daten zwischen einzelnen Requests. Für den Zugriff auf den Flash-Speicher steht jeder Action die Hash *flash* zur Verfügung, die der Getter *flash* liefert:

```
def login
  flash[:welcome] = "Herzlich Willkommen!"
end
```

Der wesentliche Unterschied zwischen Flash und Session ist, dass Flash-Daten nur für zwei direkt aufeinander folgende Requests zur Verfügung stehen. D.h., wenn Action *a* einen Wert in den Flash schreibt, hat Action *b* nur dann Zugriff auf diesen Wert, wenn *b* direkt im Anschluss an *a* ausgeführt wird. Für jede danach ausgeführte Action stehen die von *a* in den Flash geschriebenen Daten nicht mehr zur Verfügung.

Der Flash eignet sich insbesondere für die Anzeige von kurzen Hinweisen oder Fehlermeldungen. Dazu ein kleines Beispiel: Soll der Benutzer nach dem Anlegen eines neuen Projekts über den Erfolg seiner Aktion informiert werden, dann ist dies ein Fall für den Flash. In Abschnitt 5.3 haben wir beschrieben, dass Actions, die neue Objekte anlegen und speichern, nachfolgende Informationen nicht durch Anzeige einer HTML-Seite, sondern mit Hilfe eines Redirects anzeigen sollen.

Für unsere *create*-Action zum Anlegen eines neuen Projekts heißt das: die Action soll nicht den *list*-View liefern, sondern auf die Action *list* weiterleiten. Da Redirects von neuen Controller-Instanzen bearbeitet werden, stehen die in der ursprünglichen Action (*create*) erzeugten Controller-Instanzvariablen in der Ziel-Action (*list*) **nicht** zur Verfügung. Jetzt kommt der Flash ins Spiel: Die von einer Action in den Flash geschriebenen Daten stehen in der direkt im Anschluss ausgeführten Action zur Verfügung. Für unser Beispiel heißt das: Die Action *create* kann eine Information in den Flash schreiben, die in der weitergeleiteten Action *list* im zugehörigen View *list* weiter zur Verfügung steht:

```
def create
  @project = Project.new(params[:project])
  if @project.save
    flash[:notice] = "Ein Projekt wurde erzeugt"
    redirect_to :action => "list"
  else
    render :action => "new"
  end
end

def list
  @projects = Project.find(:all)
end
```

Die *list*-Action liefert den *list*-View aus, der die Flash-Daten zur Anzeige der von *create* bereitgestellten Daten nutzt:

```

<body>
  <% if flash[:notice] %>
    <div><%= flash[:notice] %></div>
  <% end %>
  ...
</body>

```

An diesem Beispiel wird auch deutlich, warum es wichtig ist, dass die Flash-Daten nur im direkten Nachfolge-Request, nicht aber darüber hinaus zur Verfügung stehen. Lädt der Benutzer die *list*-Seite ein weiteres Mal, dann ist der Hinweis verschwunden, da *flash[:notice]* beim zweiten Aufruf *nil* ist.

5.9.1 Weitere Flash-Methoden

Die Methode *now* schreibt den Wert nur für die aktuelle Action in den Flash, d.h. der Wert ist nur in der Action und im anschließend angezeigten View verfügbar:

```
flash.now[:notice] = "Nur in aktueller Action bekannt"
```

Um einen Wert für einen weiteren Request-Zyklus im Flash zu halten, steht die Methode *keep* zur Verfügung:

```
flash.keep(:notice)
```

5.10 Filter

Filter sind Methoden oder Blöcke, die vor oder nach dem Aufruf einer Controller-Action automatisch ausgeführt werden. Grundsätzlich wird zwischen Before- und After-Filtern unterschieden. Before-Filter werden durch Aufruf der Klassenmethode *before_filter* definiert und vor der jeweiligen Action ausgeführt. After-Filter werden durch Aufruf der Klassenmethode *after_filter* definiert und nach der jeweiligen Action ausgeführt. Darüber hinaus bietet Rails die Möglichkeit der Definition von Around-Filtern, die wir in Abschnitt 5.10.1 beschreiben.

Das folgende Code-Beispiel definiert für den *AdminController* einen Before-Filter, der prüft, ob der angemeldete Benutzer ein Administrator ist:

```

class AdminController < ActionController::Base
  before_filter :assert_admin_role

  private
  def assert_admin_role
    session[:person].role == Role::ADMIN
  end
end

```

Before-Filter können die Ausführung einer Action abbrechen, indem sie wie im obigen Beispiel *false* liefern oder auf eine andere Action weiterleiten.

Filter haben Zugriff auf die Request- und Response-Parameter des aktuellen Requests. Das folgende Beispiel definiert einen After-Filter, der anhand eines Request-Parameters prüft, ob die Antwort des Servers vor der Übertragung zum Client verschlüsselt werden soll:

```
class ProjectsController < ActionController::Base
  after_filter :encrypt_body

  private
    def encrypt_body
      if params['use_encryption']
        response.body = encrypt(response.body)
      end
    end

    # verschlüsselt den übergebenen String
    def encrypt text
      ...
    end

    ...
end
```

Filter werden in der Reihenfolge ihrer Deklaration ausgeführt. Das folgende Beispiel definiert zwei Before- und zwei After-Filter und zeigt die Ausführungskette beim Aufruf der *index*-Action:

```
class ProjectsController < ActionController::Base
  before_filter :b
  before_filter :a
  after_filter :d
  after_filter :c

  def index
    end
end

b => a => index => d => c
```

Abschnitt 5.10.5 beschreibt, wie Sie die Ausführungsreihenfolge von Filtern unabhängig von der Reihenfolge ihrer Deklaration ändern können.

5.10.1 Around-Filter

Around-Filter kombinieren Before- und After-Filter und eignen sich insbesondere für Anforderungen, die die Verwaltung eines Zustands zwischen der Ausführung des Before- und After-Codes erfordern. Around-Filter werden durch Klassen definiert, die die beiden Methoden *before* und *after* implementieren. Das folgende Beispiel zeigt einen Around-Filter, der die Ausführungszeiten von Actions misst und entsprechend protokolliert:

```

class BenchmarkFilter
  def before
    @start = Time.now
  end

  def after
    report_timing
  end
end

class ProjectsController < ActionController::Base
  around_filter BenchmarkFilter.new
  ...
end

```

Die Before-Methode eines Around-Filters wird ans Ende der Before-Filterliste und die After-Methode an den Anfang der After-Filterliste eines Controllers gestellt.

5.10.2 Bedingungen

Ohne Angabe zusätzlicher Parameter greift ein Filter für sämtliche Actions des Controllers. Dieses Verhalten kann durch die beiden Parameter `:except` und `:only` gesteuert werden.

Der Parameter `:except` bewirkt den Ausschluss eines Filters für bestimmte Actions. Diese Technik haben wir im Login-Beispiel unseres Hands-on-Kapitels 3 verwendet, um zu vermeiden, dass der authenticate-Filter für die Actions *login* und *authenticate* aufgerufen wird:

```

class ApplicationController < ActionController::Base
  before_filter :authenticate, :except => [:login, :sign_on]
  ...
end

```

Alternativ zum Ausschlussverfahren kann ein Filter durch Verwendung des Parameters `:only` nur für bestimmte Actions aktiviert werden:

```

class ProjectsController < ActionController::Base
  before_filter :authenticate, :only => [:new, :edit]
  ...
end

```

Im Beispiel sind sämtliche Actions des *ProjectsControllers* ohne vorherige Authentifizierung nutzbar, mit Ausnahme der beiden Actions *new* und *edit*. Ein Beispiel aus der Praxis für die Verwendung von `:only` sind die vielen Shop-Systeme im Internet: sie erlauben Benutzern das Stöbern im Shop, ohne angemeldet zu sein. Erst beim Durchführen einer Bestellung ist die Authentifizierung erforderlich.

Hinweis: Die Parameter `:except` und `:only` sind nur für Before- und After-Filter anwendbar, sie funktionieren nicht für Around-Filter.

5.10.3 Filterklassen und Inline-Filter

Neben der beschriebenen Möglichkeit, Filter durch Angabe eines Methodensymbols zu definieren, können Filter außerdem als externe Klassen oder Inline-Methoden definiert werden.

Die Verwendung von externen Klassen bietet sich für den Bau von wiederverwendbaren Filtern oder Filterbibliotheken an. Eine Filterklasse muss die Klassenmethode *filter* implementieren, die als Parameter die aktuelle Controller-Instanz erhält:

```
class EncryptionFilter
  def self.filter controller
    controller.response.body = encrypt(controller.response.body)
  end
  ...

  private
  # verschlüsselt den übergebenen String
  def encrypt text
    ...
  end
end

class ProjectsController < ActionController::Base
  after_filter EncryptionFilter
  ...
end
```

Inline-Filter werden definiert, indem der Filterdeklaration ein Block übergeben wird, der als Parameter den Controller akzeptiert:

```
class ProjectsController < ActionController::Base
  before_filter do |controller|
    controller.session[:person].firstname == "Thomas"
  end
  ...
end
```

Der Filter liefert *false*, sofern die angemeldete Person nicht Thomas heißt.

5.10.4 Filtervererbung

Filter werden in der Controller-Vererbungshierarchie nach unten vererbt. Im folgenden Beispiel erben alle Nachkommen des *ApplicationControllers* den *authenticate*-Filter:

```
class ApplicationController < ActionController::Base
  before_filter :authenticate, :except => [:login, :sign_on]

  private
  def authenticate
```

```

        unless session[:person]
          redirect_to :controller => "projects", :action => "login"
        end
      end
    end

    class ProjectsController < ApplicationController
      after_filter :encrypt_body
      ...
    end

```

5.10.5 Filterketten

Manchmal ist es erforderlich, die Standardreihenfolge von Filtern abzuwandeln. Diesem Zweck dienen die beiden Methoden *prepend_before_filter* und *prepend_after_filter*, die den angegebenen Filter an den Anfang der jeweiligen Kette stellen. Ein Anwendungsbeispiel für die Modifizierung der Filterreihenfolge ist die Umgehung des Login-Vorgangs für lokale Requests.

In Kapitel 3 haben wir den *ApplicationController* um einen *authenticate*-Filter erweitert, der die Session vor jeder Action auf einen angemeldeten Benutzer überprüft:

```

class ApplicationController < ActionController::Base
  before_filter :authenticate,
               :except => [:login, :sign_on]

  private
  def authenticate
    unless session[:person]
      redirect_to :controller => "projects", :action => "login"
    end
  end
end

```

Damit wir uns während der Entwicklungsphase eines Systems nicht ständig beim System anmelden müssen, installieren wir einen weiteren Before-Filter *local_request*. Der Filter erzeugt für lokale Requests ein Benutzer-Objekt und schreibt es in die Session, sodass der *authenticate*-Filter ein gültiges Benutzer-Objekt in der Session vorfindet. Damit der Mechanismus funktioniert, ist es wichtig, dass der *local_request*-Filter vor dem *authenticate*-Filter ausgeführt wird, und genau diesem Zweck dient die Methode *prepend_before_filter*:

```

class ProjectsController < ApplicationController
  prepend_before_filter :local_request

  private
  def local_request
    if request.env["REMOTE_HOST"] == "127.0.0.1"
      session[:person] ||= Person.new(:firstname => "Ralf")
    end
  end
end

```

```

    end
  end

```

Wäre der *local_request*-Filter durch den Aufruf der Methode *before.filter* installiert worden, würde er erst nach dem *authenticate*-Filter ausgeführt werden, und der gewünschte Effekt würde nicht eintreten.

5.11 HTTP-Authentifikation

Häufig sichern Webserver den Zugriff auf geschützte Bereiche über HTTP-Authentifikation.⁹ Dabei wird der Anwender über ein Browserfenster nach einem Login und Passwort gefragt. Rails unterstützt diese Form der Anmeldung über die Methoden *authenticate_or_request_with_http_basic*, *authenticate_with_http_basic* und *request_http_basic_authentication*. Im folgenden Beispiel nutzen wir die HTTP-Authentifikation in einem Filter:

```

class LoginController < ApplicationController
  before_filter :authenticate
  ...
  private

  def authenticate
    authenticate_or_request_with_http_basic do |login, password|
      if user = User.authenticate(login, password)
        session[:user_id] = user.id
        return true
      end
      return false
    end
  end
end

```

Die Methode *authenticate_or_request_with_http_basic* erhält Login und Passwort aus der HTTP-Authentifikation und leitet diese an den übergebenen Block weiter. In diesem wird die Prüfung auf gültige Parameter durchgeführt. Bei einem erfolgreichen Login, liefert der Block *true*, andernfalls *false*, wodurch der Filter bzw. die Filterkette abgebrochen wird (vgl. Abschnitt 5.10).

Die anderen beiden Methoden erlauben, das Verhalten des Filters etwas genauer zu steuern:

```

class LoginController < ApplicationController
  before_filter :authenticate
  ...
  private

  def authenticate
    user = authenticate_with_http_basic do |login, password|

```

⁹http://en.wikipedia.org/wiki/Basic_authentication_scheme

```
User.authenticate(login, password)
end
if user
  session[:user_id] = user
  return true
end
request_http_basic_authentication and return false
end
end
```

5.12 Routing und URL-Generierung

Das Thema Routing ist sehr umfangreich, und die vollständige Beschreibung aller Aspekte liegt außerhalb dieses Buches. Wir beschreiben in diesem Abschnitt daher nur die wichtigsten Punkte. Weitere Informationen liefert auch das Kapitel 7.

5.12.1 Routing

Der Begriff *Routing* beschreibt zwei Aufgaben von Rails; das Mapping einer URL auf den Controller und die Action und die Erzeugung von URLs bei der Angabe von Methoden wie *link_to*, *redirect_to* usw.

Standardmäßig extrahiert Rails Controller- und Action-Name sowie eventuelle weitere Parameter aus der URL und führt anschließend die entsprechende Action auf einer frischen Controller-Instanz aus. Dieses Standardverhalten basiert auf den Einstellungen in der Konfigurationsdatei *config/routes.rb*. Hier ein Auszug aus der Datei:

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id'
end
```

Die Datei *routes.rb* enthält mindestens einen *map.connect*-Aufruf, sie kann aber auch mehrere enthalten. Jeder Aufruf *map.connect* definiert eine Route von einer eingehenden URL auf eine Controller-Action. Der *connect*-Aufruf akzeptiert dafür ein Muster, gefolgt von einer Reihe optionaler Parameter. Das Muster besteht aus einer beliebigen Zahl einzelner, durch „/“ getrennter Komponenten.

Der Routing-Vorgang bildet jeden Teil der eingehenden URL auf eine Komponente im Muster ab. So definiert der im Listing dargestellte *connect*-Aufruf genau das *Controller-Action-Id*-Muster, welches für die standardmäßige Abbildung der URL auf eine Controller-Action verwendet wird.

Abbildung 5.5 veranschaulicht den Rails Routing-Prozess am Beispiel der URL *http://localhost:3000/projects/show/1*. Die URL ist der Eingabeparameter des Routings. Der Routing-Prozess parst die URL ab der ersten Stelle hinter dem Hostnamen und ggf. Port. Als Erstes findet der Prozess das Wort *projects* und weiß auf Grund seiner Routing-Konfiguration *routes.rb*, dass es sich hierbei um den Controller, in diesem Fall also den *ProjectsController* handelt. Als Nächstes wird das Wort

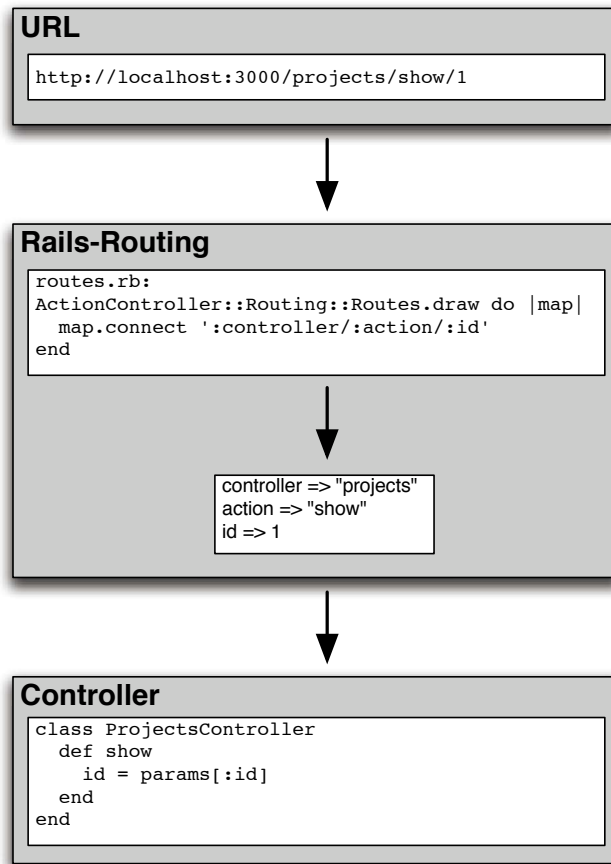


Abbildung 5.5: Der Rails Routing-Prozess

`show` gefunden, was der Konfiguration der Action entspricht. An dritter und letzter Stelle findet der Routing-Prozess die Zahl 1, die gemäß der Konfiguration der Request-Parameter mit dem Bezeichner `id` ist.

Intern erzeugt der Routing-Prozess für jeden Request eine Hash `params` mit den ermittelten Parametern. Für die URL `http://localhost:3000/projects/show/1` sieht die Hash wie folgt aus:

```
params = {:controller => "projects",
          :action => "show",
          :id => 1}
```

Nachdem die URL analysiert und die Hash gefüllt wurden, erzeugt Rails eine neue Instanz des in dieser Hash unter dem Schlüssel `:controller` angegebenen Controllers und ruft deren Methode `process` auf, die den Aufruf an die eigentliche Ziel-Action `show` delegiert.

Abbildung 5.5 macht deutlich, dass jede Komponente der eingehenden URL auf eine Komponente des dem `connect`-Aufruf übergebenen Musters abgebildet wird. Eine

besondere Rolle im Muster kommt den beiden Komponenten `:controller` und `:action` zu, da diese beiden Parameter für die Ermittlung und Ausführung der Action zwingend nötig sind. Fehlen eine oder beide dieser Komponenten, müssen sie als zusätzliche Parameter des `connect`-Aufrufs angegeben werden:

```
map.connect ':action/:id'
```

Das Muster bewirkt, dass Rails aus der URL `http://localhost:3000/show/1` die folgende Parameter-Hash erzeugt:

```
params = {:action => "show",  
          :id => 1}
```

Jetzt haben wir ein Problem: Auf Grund des fehlenden Controllernamens in der Hash weiß Rails nicht, welcher Typ von Controller für die Ausführung der `show`-Action zuständig ist. Die Lösung des Problems ist die Angabe von optionalen Parametern, mit denen wir die Controllerklasse explizit spezifizieren können:

```
map.connect ':action/:id', :controller => 'projects'
```

Dieser Eintrag bewirkt, dass alle HTTP-Requests vom `ProjectsController` bearbeitet werden und die resultierende `params`-Hash wie folgt aussieht:

```
params = {:controller => "projects",  
          :action => "show",  
          :id => 1}
```

Das Spiel mit den optionalen `connect`-Parametern können Sie beliebig weitertreiben: Jeder optionale Parameter taucht in der `params`-Hash unter dem von Ihnen angegebenen Schlüssel auf und steht damit der Ziel-Action als Parameter zur Verfügung.

Die Datei `config/routes.rb` kann mehrere `connect`-Aufrufe enthalten, deren Reihenfolge die Priorität der jeweiligen Route vorgibt. Rails geht die Datei also von oben nach unten durch und wendet die erste passende Regel auf die umzusetzende URL an.

5.12.2 Anpassung des Routings

Sie können sich durch Anpassung der Datei `routes.rb` Ihr eigenes Routing definieren. Ein typischer Fall ist z.B. die Anforderung, dass alle URLs einer Applikation mit dem Namen der Applikation beginnen sollen. Für unsere Projektmanagement-Software sollen beispielsweise alle URLs mit dem Namen `ontrack` beginnen. Das gewünschte Verhalten wird durch die folgende Routing-Konfiguration `routes.rb` definiert:

```
map.connect 'ontrack/:controller/:action/:id'
```

Die URL `http://localhost:3000/ontrack/projects/show/1` resultiert in folgender Parameter-Hash:

```
params = {:controller => "projects", :action => "show", :id => 1}
```

und führt damit zum Aufruf der Action `show` auf dem `ProjectsController` mit der ID 1 für die zu ladende Modellinstanz.

Als Nächstes benötigen wir einen Routing-Eintrag, der die Startseite der Anwendung festlegt. Gibt der Benutzer beispielsweise die URL `http://localhost:3000/ontrack` ohne weitere Parameter ein, dann soll er automatisch auf die Login-Seite der Anwendung gelangen. Der folgende *connect*-Aufruf sorgt für das gewünschte Verhalten:

```
map.connect 'ontrack', :controller => "projects",
              :action => "login"
```

Der Eintrag gibt für URLs, die nach dem Hostnamen nur noch das Wort *ontrack* enthalten, sowohl den Controller als auch die zu aktivierende Action vor.

Damit die Seiten der Anwendung nicht mehr ohne das Präfix *ontrack* geöffnet werden können, müssen wir noch den Standardeintrag aus *routes.rb* löschen. Das folgende Listing zeigt die nun vollständige Routing-Konfiguration:

Listing 5.1: config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.connect 'ontrack/:controller/:action/:id'
  map.connect 'ontrack', :controller => "projects",
                        :action => "login"
  map.connect('/:controller/:action/:id')
end
```

5.12.3 Root Route

Wenn Sie in der URL keinen Controller und keine Action angeben, wird die Rails-Startseite *public/index.html* angezeigt. Wenn Sie stattdessen eine Seite aus der Anwendung anzeigen möchten, definieren Sie diese als so genannte *Root Route*:

Listing 5.2: config/routes.rb

```
# http://localhost:3000/
map.root :controller => "projects", :action => "login"
```

Sie müssen zusätzlich die Datei *public/index.html* löschen, da sie aufgrund der Rewrite-Rule in *public/.htaccess* angezeigt wird.

5.12.4 Named Routes

Named Routes wurden eingeführt, um dem Programmierer die Arbeit zu erleichtern und den Quellcode noch lesbarer zu machen. Statt in allen Controllern und Views, d.h. bei *redirect_to*, *link_to* usw eine Hash aus Controller und Action anzugeben, verwenden Sie besser die so genannten *Named Routes*:

```
def logout
  reset_session
  redirect_to { :controller => 'login', :action => 'login' }
  redirect_to login_url
end
```

```
link_to 'Login', {:controller => 'login', :action => 'login'}  
link_to 'Login', login_path
```

Bei den Named Routes handelt es sich um Methoden, die Ihnen Rails bereitstellt, wenn Sie in der Datei *config/routes.rb* entsprechende Einträge definieren. Die Routes für *login* und *logout* sehen als Beispiel wie folgt aus:

Listing 5.3: *config/routes.rb*

```
map.login 'anmelden', :controller => "login", :action => "login"  
map.logout 'abmelden', :controller => "login", :action => "logout"
```

Der Name der Route *login* wird durch den Aufruf von *login* auf der Instanz *map* definiert. Daraus erzeugt Rails die Methoden *login_path* und *login_url*, die Sie überall in Ihrem Code nutzen können. Die Methode mit der Endung *_path* erzeugt dabei einen relativen Pfad, die Methode mit der Endung *_url* den absoluten:

```
login_path => /anmelden  
login_url  => http://localhost:3000/anmelden
```

Wie Sie sehen, trägt Rails hier nicht den Controller und die Action ein (*/login/login*), sondern den Namen, der über den ersten Parameter zur Route definiert wurde (hier „anmelden“). Sie können hier auch den Namen aus Controller und Action angeben:

```
map.login 'login/login', :controller => "login", :action => "login"
```

Durch die als zweiter Parameter angegebene Hash mit Controller und Action und ggf. weiteren Argumenten (z.B. *:id*) wird festgelegt, welcher Controller und welche Action für die Verarbeitung des Requests zuständig sind. Der Aufruf der URL *http://localhost:3000/anmelden* wird im Beispiel somit vom Controller *login* und der Action *login* verarbeitet.

Die Definition mehrerer Routes zu einem Controller kann durch die Methode *with_options* vereinfacht werden:

Listing 5.4: *config/routes.rb*

```
map.with_options :controller => 'login' do |auth|  
  auth.login 'anmelden', :action => "login"  
  auth.logout 'abmelden', :action => "logout"  
end
```

Verwenden Sie keine Named Routes und ändern später den Controller- oder den Action-Namen, so müssen Sie alle Controller und Views nach dem Vorkommen des Controller- oder Action-Namens durchsuchen und anpassen. Verwenden Sie stattdessen Named Routes, so müssen Sie nur die Definition der Route in *config/routes.rb* anpassen:

Listing 5.5: *config/routes.rb*

```
# alt  
map.with_options :controller => 'login' do |auth| ...  
# neu  
map.with_options :controller => 'authenticate' do |auth| ...
```


Named Routes sind für das Thema REST und Rails von besonderer Bedeutung. Wir gehen im Abschnitt 7.10 näher darauf ein und zeigen Beispiele für Routes mit Parametern und Routes für Modellassoziationen. Die Informationen sind auch für Anwendungen hilfreich, die nicht REST-basiert sind. Sie sollten sich in jedem Fall angewöhnen, ausschließlich Named Routes zu verwenden.

5.12.5 URL-Generierung

Neben dem Mapping einer URL auf Controller und Action ist das Routing für den umgekehrten Weg, nämlich für die Erzeugung einer URL aus Controller- und Action-Angabe zuständig. Die Seiten einer Applikation werden nicht nur durch die Eingabe von URLs geöffnet, sondern jede Applikation enthält auch Links und Buttons, die auf andere Seiten der Anwendung verweisen. Das folgende Code-Fragment enthält z.B. den Link *Zurück* auf die Übersichtsseite mit der Projektliste:¹⁰

```
<a href="/projects/list">Zurück</a>
```

Angenommen, wir hätten unser Routing so konfiguriert, dass jede URL unserer Anwendung mit dem Applikationsnamen *ontrack* beginnt, dann würde der oben dargestellte Link nicht mehr funktionieren. Das Routing würde für die URL */projects/-list* keinen gültigen Eintrag finden und in einem Routing-Error resultieren (siehe Abbildung 5.6).



Abbildung 5.6: Das Routing schlägt fehl.

Das eigentliche Problem ist, dass der Applikationscode nur noch die vom Routing erzeugten Request-Parameter, wie *controller* oder *action*, sieht, doch keine Information über die ursprüngliche URL */ontrack/projects/list* mehr besitzt. Wir können unsere Links in Controllern und Views somit nur auf Basis der zur Verfügung stehenden Parameter codieren.

Die Rails-Lösung dieses Problems heißt *url_for*. Die Methode steht in Views und Controllern zur Verfügung und ist für die Umsetzung von Request-Parametern in korrekte URLs zuständig:

```
link = url_for :controller => "projects", :action => "list"
```

Der Aufruf resultiert in einer korrekten URL, die wiederum vom zugrunde liegenden Routing in einen korrekten Request übersetzt werden kann:

¹⁰ Wir verwenden hier bewusst das HTML-Tag `<a>` und nicht den Formular-Helper `link_to`, da auf diese Weise das Prinzip der URL-Generierung besser verständlich wird.

```
http://localhost:3000/ontrack/projects/list
```

Fehlende Parameter versucht *url_for* mit Werten des aktuellen Requests zu ersetzen. Wird z.B. der *:controller*-Parameter weggelassen, dann füllt Rails den Parameter mit dem Namen des aktuellen Controllers auf.

Die Methode *url_for* wird von diversen High-Level-Funktionen, wie z.B. *link_to* oder *redirect_to*, verwendet. Das heißt, wenn wir für den zu Beginn dieses Abschnitts programmierten Link anstelle des HTML-Tags `<a>` den Formular-Helfer *link_to* verwenden, tritt das beschriebene Problem gar nicht erst auf, und das Routing funktioniert in beide Richtungen korrekt.

5.13 Layout

Controller sind nicht nur für die Steuerung des Kontrollflusses einer Anwendung zuständig, sondern bestimmen bis zu einem gewissen Grad auch deren Aussehen durch Vorgabe eines oder mehrerer Layouts. Ein Layout ist eine RHTML-Datei, die als eine Art Wrapper für den eigentlichen Inhalt der Seite fungiert. Sie können sich ein Layout als HTML-Datei mit Header und Footer vorstellen, die an einer bestimmten Stelle den Platzhalter *yield* für den Inhalt der Seite enthält:

```
<h1>Kopfzeile der Anwendung</h1>
<%= yield %>
<h1>Fusszeile der Anwendung</h1>
```

Die Layoutdateien eines Projekts liegen im Verzeichnis *app/views/layouts* (siehe Abbildung 5.7).

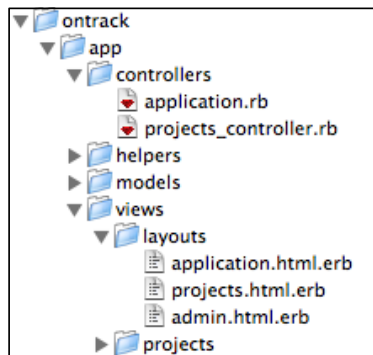


Abbildung 5.7: Eine Rails-Anwendung kann mehrere Layouts besitzen.

Die Einbindung von Layouts erfolgt über Controllerklassen. Im Normalfall gilt ein Layout für einen Controller oder sogar für die gesamte Anwendung. In Einzelfällen kann es sinnvoll sein, für bestimmte Actions vom Normalfall abzuweichen und sie mit einem Action-spezifischen Layout darzustellen. Im Folgenden werden die verschiedenen Konzepte für die Zuweisung von Layouts beschrieben.

5.13.1 Automatische Layoutzuweisung

Rails weist einem Controller automatisch ein Layout zu, wenn sich im Layoutverzeichnis eine dem Controllernamen entsprechende Layoutdatei befindet und der Controller keine explizite Layoutzuweisung enthält. Beispielsweise würde dem *ProjectsController* automatisch das Layout *projects.html.erb* zugewiesen, wenn die genannten Voraussetzungen erfüllt sind.

Befindet sich im Layoutverzeichnis keine dem Controllernamen entsprechende Layoutdatei, aber eine Layoutdatei mit dem Namen *application.html.erb*, dann macht Rails diese Datei zum Standard-Layout des Controllers. Auch wieder nur unter der Voraussetzung, dass der Controller keine explizite Layoutzuweisung enthält.

Vererbung

Automatisch zugewiesene Layouts werden vererbt. Der folgende Codeausschnitt demonstriert das Prinzip am Beispiel der Controller *Ontrack* und *Projects*:

```
class OntrackController < ActionController::Base
end

class ProjectsController < OntrackController
end
```

Angenommen, im Layoutverzeichnis befindet sich ein Layout mit dem Namen *ontrack*, dann erhält neben dem Controller *Ontrack* zusätzlich der Controller *Projects* das Layout *ontrack* zugewiesen, sofern sich kein Layout mit dem Namen *projects* im Layoutverzeichnis befindet.

Erbt ein Controller ein Layout, das er gar nicht will, so kann das Layout durch den Aufruf *layout nil* in der erbenden Klasse deaktiviert werden:

```
class ProjectsController < OnTrackController
  layout nil
end
```

5.13.2 Explizite Layoutzuweisung

Die explizite Layoutzuweisung erfolgt über den Aufruf der statischen Methode *layout*:

```
class ProjectsController < ActionController::Base
  layout "ontrack"
end
```

Vererbung

Explizit zugewiesene Layouts werden vererbt. Im folgenden Beispiel erbt der *ProjectsController* das Layout *ontrack* vom *OntrackController*. Der *AdminController* überschreibt das geerbte Layout mit dem *admin*-Layout.

```
class OntrackController < ActionController::Base
  layout "ontrack"
end

class ProjectsController < OntrackController
end

class AdminController < OntrackController
  layout "admin"
end
```

Für die explizite Layoutzuweisung gilt genau wie für die automatische: Erbt ein Controller ein Layout, das er gar nicht will, so kann das Layout durch den Aufruf *layout nil* in der erbenden Klasse deaktiviert werden.

5.13.3 Dynamische Bestimmung des Layouts

Häufig wird ein und dieselbe Web-Anwendung von verschiedenen Kunden bzw. Firmen genutzt. Jede dieser Firmen hat eine bestimmte „Corporate Identity“, die nicht nur auf Visitenkarten oder Briefpapier zu erkennen ist, sondern auch am Web-auftritt der Firma.

Die Anforderung, dass sich das Layout einer Anwendung in Abhängigkeit des jeweils angemeldeten Benutzers ändern soll, ist daher nicht ungewöhnlich und lässt sich in Rails-Anwendungen mit Hilfe dynamischer Layouts umsetzen. Die dynamische Bestimmung des Layouts erfolgt, indem der *layout*-Methode anstelle des Layoutnamens ein Methodensymbol bzw. eine Inline-Methode übergeben wird:

```
class ProjectsController < ActionController::Base
  layout :determine_layout

  private

  def determine_layout
    session[:user].name == "Ralf" ? "special" : "normal"
  end
end
```

Die von *layout* referenzierte Methode ermittelt dann in Abhängigkeit von bestimmten Kriterien (z.B. dem aktuell angemeldeten Benutzer) das zu verwendende Layout und liefert dessen Namen als String.

5.13.4 Action-spezifische Layouts

In einigen Fällen ist es erforderlich, für eine oder mehrere Actions vom Layout des jeweiligen Controllers abzuweichen. Angenommen, der *ProjectsController* besitzt eine Action, die die Liste der Tasks einer Iteration als RSS-Feed¹¹ liefert. In diesem Fall ist es unsinnig bzw. gar nicht möglich, den RSS-Stream mit einem Layout auszu-

¹¹<http://de.wikipedia.org/wiki/RSS>

statten. Sie können das Layout für einzelne Actions mit Hilfe des Parameters *:except* der Methode *layout* unterdrücken. Die folgende Layoutdeklaration sorgt dafür, dass die Antwort der *RSS*-Action ohne Layout an den Client geliefert wird:

```
class ProjectsController < ActionController::Base
  layout "ontrack", :except => :rss
  ...
end
```

In einigen Fällen ist das Unterdrücken eines Layouts nicht ausreichend. Statt des Standardlayouts soll ein Action-spezifisches Layout verwendet werden. In diesem Fall ist der Parameter *layout* des *render*-Aufrufs zu nutzen:

```
class ProjectsController < ActionController::Base
  def about
    render :action => "about", :layout => "bsimple"
  end
end
```

Weitere Informationen zu Layouts finden Sie auch im Abschnitt 6.4.

5.14 Datei-Download

Controller können Dateien an ihre Clients senden. Hierfür stehen die Methoden *send_data* und *send_file* zur Verfügung.

5.14.1 Die Methode *send_data*

Die Methode *send_data* sendet einen String mit Binärdaten an den Client.

```
send_data(data,
  :filename => 'b-simple.gif',
  :type => 'image/gif',
  :disposition => 'inline')
```

data

Daten, die an den Client übertragen werden sollen.

:filename

Dateiname, den der Browser als Vorschlag benutzt.

:type

Gibt den HTTP-Content-Typ der Datei an. Der Defaultwert ist: *application/octet-stream*.

:disposition

Gibt an, ob der Browser die Datei intern, d.h. innerhalb des Browsers, oder extern anzeigen soll. Mögliche Werte sind *inline* und *attachment* (default).

5.14.2 Die Methode `send_file`

Die Methode `send_file` sendet eine Datei in Blöcken von standardmäßig 4096 Bytes an den Client, sofern der Webserver dies unterstützt. Die Datei muss dann nicht vollständig in den Speicher geladen werden, bevor die Übertragung an den Client beginnen kann.

```
send_file('/public/b-simple.gif',
          :type => 'image/gif',
          :disposition => 'inline',
          :stream => true,
          :buffer_size => 4096)
```

filename

Name (inkl. Pfad) der zu übertragenden Datei.

:type

Gibt den HTTP-Content-Typ der Datei an. Der Defaultwert ist: *application/octet-stream*.

:disposition

Gibt an, ob der Browser die Datei intern, d.h. innerhalb des Browsers, oder extern anzeigen soll. Mögliche Werte sind *inline* und *attachment* (default).

:stream

Gibt an, ob die Datei schon während des Lesens oder erst nachdem sie vollständig gelesen wurde, an den Client gesendet werden soll. Mögliche Werte sind *false* und *true* (default).

:buffer_size

Blockgröße in Bytes, die für die Übertragung der Datei an den Client verwendet wird. Der Defaultwert ist 4096 Bytes.

5.14.3 Unsichere Dateidownloads verhindern

Die Methode `send_file` aus Abschnitt 5.14.2 erwartet als Parameter den Namen (inkl. Pfad) der zu übertragenden Datei. Wird der Dateiname direkt und ohne weitere Prüfung aus den Request-Parametern an die Methode übergeben, können Angreifer möglicherweise beliebige Dateien vom Server herunterladen:

```
def download
  send_file(params[:path])
end
```

```
http://meinedomain.de/reports/download?path=/geheim/intern.pdf
```

Abgesehen davon, dass Sie das Dateisystem mit entsprechenden Zugriffsrechten versehen, sollten Request-Parameter auf keinen Fall ohne vorherige Prüfung an die Methode `send_file` übergeben werden.

5.15 Datei-Upload

Viele Web-Anwendungen beinhalten Funktionen für den Up- und Download von Dateianhängen. Ein Beispiel sind Web-basierte E-Mail-Anwendungen: Benutzer können zu versendende E-Mails mit Dateianhängen versehen, aber auch E-Mails mit Anhängen empfangen und die Anhänge im Browser bzw. in einer externen Anwendung anzeigen. Sie können diese Funktionalität selbst entwickeln, aber wir empfehlen Ihnen dafür das Plugin *attachment_fu* unter der URL http://svn.technoweenie.net/projects/plugins/attachment_fu/.

5.16 Rails-Konsole für Controller nutzen

Die in Abschnitt 4.3 beschriebene Rails-Konsole kann auch für Tests bezüglich Controllern und Actions dienen. Wir zeigen anhand eines kurzen Beispiels, wie eine solche Konsolen-Sitzung aussehen kann:

```
ontrack $ script/console
Loading development environment (Rails 2.0.2)
01 >> app.get '/projects'
02 => 302
03 >> app.controller.params
04 => {"action"=>"index", "controller"=>"projects"}
05 >> app.response.redirect_url
06 => "http://www.example.com/login"
07 >> app.post '/authentication/sign_on', {:username => 'ontrack',
      :password => 'ontrack'}
08 => 422
09 >> app.get '/authentication/login'
10 => 200
11 >> app.response.body.grep(/authenticity_token/)
12 => ...<input name="authenticity_token"
      value="\87ffb3424bb70297f621d83eblaca5cedbddf5ee\" ...
13 >> app.post '/authentication/sign_on', {:username => 'ontrack',
      :password => 'ontrack',
      :authenticity_token => '87ffb3424bb70297f621d83eblaca...
14 eblaca5cedbddf5ee'}
15 => 302
16 >> p app.session[:person]
17 1
18 => nil
19 >> app.get '/projects'
20 => 200
21 >> app.response.body
22 >> ...
23 >> quit
```

Nach dem Aufruf der Konsole verwenden wir die Instanz *app* vom Typ *ActionController::Integration::Session*, um auf die Actions unserer OnTrack-Anwendung

zuzugreifen. Hierbei handelt es sich um eine Instanz, die eine Sitzung mit der Anwendung definiert.

In Zeile 01 führen wir einen GET-Request auf die Projektliste durch, der den Rückgabewert 302 für *Moved Temporarily* liefert. Statt des erwarteten Rückgabewerts 200 für OK wird also eine Weiterleitung erzeugt. Wir prüfen den Aufruf und lassen uns in Zeile 03 die Request-Parameter zeigen, die aber auf die richtige Action zeigt. In Zeile 05 lassen wir uns daher die Seite anzeigen, auf die die Weiterleitung erfolgt. Wir sehen, dass es sich um die Login-Seite handelt. Nun ist klar, dass wir die Projektliste nicht aufrufen können, denn wir sind noch nicht angemeldet. Mithilfe des POST-Requests und des Parameters für *username* und *password* versuchen wir ein Login, das allerdings mit dem Returnwert 422 scheitert. Ein Blick in die Log-Datei *log/development.log* zeigt, dass die Ausnahme *ActionController::InvalidAuthenticityToken* geworfen wird, da wir kein *Authenticity Token* mitliefern (vgl. Abschnitt 6.3.2).

Das aktuell gültige Token ermitteln wir, indem wir uns die Login-Seite mit dem Anmeldeformular in Zeile 09 anzeigen lassen und nach dem dort eingefügten Parameter *authenticity_token* filtern. Den Wert geben wir nun beim erneuten Anmeldeversuch in Zeile 13 an. Das Ergebnis ist eine erfolgreiche Anmeldung, die durch eine Weiterleitung angezeigt wird. In Zeile 16 stellen wir nochmals sicher, dass nun eine Benutzer-ID in der Session vorhanden ist. Der erneute Aufruf der URL */projects* liefert nun den Rückgabewert 200, und über *app.response.body* können wir uns den gesamten View ausgeben lassen.

Auf diese Weise können Sie neben dem Browser das Verhalten Ihrer Anwendung auch über die Rails-Konsole testen. Für weitere Informationen werfen Sie einen Blick in die Online-Dokumentation.

Kapitel 6

Action View

Das Sub-Framework *Action View* repräsentiert das *V* in der MVC-Architektur von Rails. Es enthält alles, was Rails zur Erstellung von Views anbietet. Der View repräsentiert die Sicht auf die Daten und wird nach der Verarbeitung eines Requests zur Anzeige an den Client zurückgesandt. Im Prinzip kann ein View jedes Format annehmen, typischerweise handelt es sich aber um HTML, XML oder JavaScript.

6.1 HTML-Templates

Ein HTML-Template trägt die Endung *.html.erb*.¹ Alle erstellten und von Rails verwendeten Templates liegen per Default im Verzeichnis *app/views* relativ zum Hauptverzeichnis des Projekts.

Zu einem Controller gibt es in der Regel mehrere Views, denn jeder Controller definiert typischerweise mehrere Actions, zu denen es jeweils einen View gibt. Unterhalb von *app/views* existiert zu jedem Controller ein Verzeichnis mit dem Namen des Controllers, in das die Views, d.h. die Templates, abgelegt werden. Ein Template *index.html.erb* zum Controller *ProjectsController* erstellen Sie somit unter *app/views/projects/index.html.erb*.

Das Template selbst trägt häufig den Namen der Controller-Action, die es verwendet. In diesem Fall findet Rails das Template am Ende einer Action automatisch. Wenn Sie für Ihr Template einen anderen Namen als den der Controller-Action wählen, müssen Sie diesen in der Action explizit angeben (vgl. Abschnitt 5.1.2).

Im einfachsten Fall enthält ein Template nur HTML-Code, der unverändert an den Browser zurückgeliefert wird.

```
<html>
  <h1>Hallo Rails!</h1>
</html>
```

¹ Die Endung *.rhtml* ist ebenfalls möglich, wird aber nicht bevorzugt.

In den meisten Fällen enthält das Template neben HTML auch Ruby-Code, der für das dynamische Einfügen von Daten sorgt. Hierbei handelt es sich um so genanntes *Embedded Ruby*, kurz *ERB*, und daher auch die Dateierweiterung *html.erb*. Embedded Ruby ist Teil der Sprache Ruby und wird über die Klasse *ERB* aus der Standard-Bibliothek bereitgestellt und von Rails genutzt. Sie müssen somit keine weitere Template-Sprache erlernen und können alle Features von Ruby auch in Ihren Templates verwenden. Alternativ können Sie aber auch andere Template-Sprachen wie z.B. *HAML*² nutzen, deren Endung dann *html.haml* lautet.

Wie in folgendem Beispiel zu sehen ist, werden Kontrollanweisungen in `<%` und `%>` eingeschlossen:

```
<h1>Aktuelle Projekte:</h1>
<%# Liste aller Projekte -%>
<ul>
  <% @projects.each do |prj| %>
    <li><%= prj.name %></li>
  <% end %>
</ul>
```

Dieser Ruby-Code erscheint nicht in HTML, hinterlässt aber immer einen Zeilenumbruch, wenn nach dem schließenden Tag `%>` kein weiterer Text kommt. Um dies bei Bedarf zu vermeiden, ist die Anweisung mit `-%>` abzuschließen.

Das Ergebnis des Ruby-Codes wird in HTML eingefügt, wenn der Ausdruck in `<%=` und `%>` eingebettet ist. Der ausgewertete Ausdruck wird hierbei mit Hilfe der Methode `to_s` in einen String gewandelt und ins HTML eingefügt. Die Anweisungen und Ausdrücke können dabei über mehrere Zeilen gehen.

Der obige Code sieht nach der Verarbeitung durch den ERB-Parser und unter Verwendung von `<%=` und `-%>` wie folgt aus:

```
<h1>Aktuelle Projekte:</h1>
<ul>
  <li>Mein erstes Ruby Projekt</li>
  <li>OnTrack Rails-Projekt</li>
  <li>Pinguin-Projekt</li>
</ul>
```

In `<%#` und `%>` eingeschlossener Text wird als Kommentar interpretiert. Dabei darf der Text keine ERB-Tag wie `<%` enthalten. Um mehrere Zeilen mit ERB-Tag auszukommentieren, verwenden Sie am besten eine If-Anweisung wie folgt:

```
<h1>Aktuelle Projekte:</h1>
<% if false -%>
<ul>
  <% @projects.each do |prj| %>
    <li><%= prj.name %></li>
  <% end %>
</ul>
<% end -%>
```

²<http://haml.hamptoncatlin.com/>

Ein Auskommentieren über die HTML-Kommentarzeichen ist möglich, doch wird der ERB-Parser den dazwischen enthaltenen Ruby-Code dennoch ausführen. Kommt es dabei zu einem Fehler, wird die Erzeugung der Seite abgebrochen:

```
<h1>Aktuelle Projekte:</h1>
<!--
  Dieser Ruby-Code wird ausgeführt!
  <% @projects.each do |prj| %>
  <% end %>
-->
...
```

In jedem View stehen Ihnen alle in der Controller-Action zum View definierten Instanzvariablen (z.B. *@project*) zur Verfügung. Wie in Abschnitt 5.2.1 beschrieben, gelangen auf diese Weise Daten aus dem Controller in den View. Wenn nötig, stehen auch die Controller-Attribute *session*, *params*, *flash*, *request*, *response* und *headers* bereit. Ebenso das aktuelle Controller-Objekt über *controller* und das Template-Verzeichnis über *base_path*.

6.2 Helper-Module

Im Sinne von MVC und einer klaren Trennung der Verantwortlichkeiten sollte ein Template nicht zu viel Ruby-Code und vor allem keine Geschäftslogik enthalten. Andernfalls wird die Wartbarkeit aufgrund der Abhängigkeiten zwischen den einzelnen Layern Model, View und Controller deutlich erschwert. Außerdem macht eine klare Trennung von View und Logik den View übersichtlicher.

Um das HTML frei von zu viel Ruby-Code zu halten, sollten Sie komplexere Code-Teile in Methoden auslagern. Rails stellt Ihnen dazu pro Controller ein Helper-Modul zur Verfügung, welches bei der Generierung des Controllers erzeugt wird. Dieses Modul ist für die Definition von Hilfsmethoden für die Templates des Controllers vorgesehen. Alle hier definierten Methoden stehen in allen Templates dieses Controllers automatisch zur Verfügung.

Im nächsten Beispiel definiert das Modul *ProjectsHelper* die folgende Helper-Methode *project_name_with_link*:

Listing 6.1: app/helpers/project_helper.rb

```
module ProjectsHelper
  def project_name_with_link name
    ...
  end
end
```

Diese ist ohne weitere Anpassungen in jedem View des *ProjectsController* verwendbar:

```
<html>
...
<%= project_name_with_link @project.name %>
```

```
...
</html>
```

Alle Helper-Module sind unter dem Verzeichnis *app/helpers* abgelegt. Per Konvention erhält eine solche Datei den Namen des Controllers und das Suffix *_helper*. Der Modulname selbst besteht aus dem Namen des Controllers, gefolgt vom Suffix *Helper*. Der Controller namens *ProjectsController* sucht somit z.B. nach einem Modul *ProjectsHelper* in der Datei *app/helpers/project_helper.rb*.

Helper-Module enthalten Methoden, die für die Templates eines Controllers spezifisch sind. Daher sind Methoden aus dem Helper-Modul zum *TasksController* nicht in Templates zum *ProjectsController* sichtbar. Um diese Methoden dennoch nutzen zu können, muss das Helper-Modul explizit über die Klassenmethode *helper* in den Controller eingebunden werden:

Listing 6.2: *app/controllers/application.rb*

```
class ProjectsController < ApplicationController
  # Helper aus helpers/task_helper.rb stehen in Views
  # zum ProjectsController zur Verfügung
  helper :task
  ...
end
```

Da aber häufig mehrere Helper in unterschiedlichen Controllern notwendig sind, bindet Rails per Default gleich alle Helper über den *ApplicationController* ein. Da es sich um den Basis-Controller aller Controller handelt, stehen Ihnen alle Helper in allen Views zur Verfügung. Wollen Sie dies nicht, müssen Sie die Zeile löschen:

Listing 6.3: *app/controllers/application.rb*

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time
  ...
end
```

Darüber hinaus können Sie allgemein gültige Methoden über den *ApplicationHelper* unter *helpers/application_helper.rb* bereitstellen. Alle hier definierten Methoden stehen in allen Templates aller Controller zur Verfügung. Dies führt aber häufig dazu, dass dieses Modul mit Hilfsmethoden überfrachtet und unübersichtlich wird.

Besser ist, Sie teilen logisch zusammenhängende Methoden in einzelne Helper-Module wie *TextFormattingHelper* oder *AuthenticationHelper* auf und binden diese über *:all* oder durch explizite Angabe im *ApplicationController* ein:

Listing 6.4: *app/controllers/application.rb*

```
class ApplicationController < ActionController::Base
  #helper :all
  helper :text_formatting, :authentication
  ...
end
```

6.3 Action View Helper

Action View bietet bereits eine Fülle von Helper-Methoden. Diese umfassen das Formatieren von Zahlen und Strings, die Erzeugung von Formular-Elementen, HTML-Tags und JavaScript. Der HTML-Code wird dadurch kürzer, sauberer und besser wartbar. Bevor Sie beginnen, eigene Hilfsmethoden zu schreiben, werfen Sie einmal einen Blick in die Online-Dokumentation zu *Action View*.

Die konkreten Module befinden sich alle unter dem Namensraum von *ActionView::Helpers* (z.B. *ActionView::Helpers::UrlHelper*). Sie müssen sich ihre Namen nicht merken, geschweige denn die Module explizit einbinden. Das übernimmt Rails für Sie und stellt Ihnen alle Methoden direkt zur Verfügung.

Einige der Helper-Methoden, wie z.B. *link_to*, haben Sie bereits in Kapitel 3 kennengelernt. In den folgenden Abschnitten stellen wir Ihnen exemplarisch einige weitere Helper-Methoden vor. Die komplette Referenz entnehmen Sie bitte der Online-Dokumentation zu *Action View*.

6.3.1 Formulare

form_for(record_or_name_or_array, *args, &proc)

Die Methode *form_for* erzeugt ein HTML-Formular für ein Modell:

```
<% form_for :project, @project,
           :url => { :action => "create" } do |f| %>
  <%= f.text_field :name %>
  <%= f.text_area :description %>
  ...
<% end %>
=>
  <form action="/projects/create" method="post">
    <input id="project_name" name="project[name]" size="30"
      type="text" value="CMS-Project" />
    ...
  </form>
```

Die Methode erwartet einen Block, in dem die *FormHelper* wie *text_field*, *text_area* angegeben werden. Diese modellbezogenen Methoden erwarten als ersten Parameter das Modell und im zweiten das Modellattribut (vgl. Abschnitt 6.3.3), z.B.:

```
<%= text_field :project, :name %>
```

Im Falle von *form_for* erfolgen die Aufrufe der Methoden auf dem Blockparameter *f*, der einen *FormBuilder* definiert. Dieser leitet die Modellinstanz (im Beispiel *@project*) an die Methoden weiter, so dass die explizite Übergabe des Modellnamens entfällt:

```
<%= f.text_field :name %>
```

Der Name der Modellinstanz ist frei wählbar und könnte im Beispiel auch *@cms-project* heißen. Der erste Parameter *:project* ist optional und ebenfalls frei

wählbar. Geben Sie den Parameter nicht an, wird dieser aus dem Klassennamen der Modellinstanz ermittelt, im Beispiel also *:project* aus *Project*.

Der Name definiert den Schlüssel, unter dem Sie die Formularwerte aus den Request-Parametern ermitteln können. Im Beispiel greifen Sie in der Methode *create* über *:project* auf die Werte zu:

```
class ProjectsController < ApplicationController
  def create
    @project = Project.new params[:project]
    ...
  end
end
```

Innerhalb des Blocks können Sie alle *FormHelper* oder *FormTagHelper* auch ohne Bezug zum Blockparameter nutzen:

```
<% form_for :project, @project,
           :url => { :action => "create" } do |f| %>
  <%= f.text_field :name %>
  <%= text_area :project, :description %>
  <%= check_box_tag :watch %>
  ...
<% end %>
```

Beachten Sie, dass *form_for* innerhalb eines ERb-Blocks (*<% %>*) und nicht in einem ERb-Ausgabeblock (*<%= %>*) verwendet wird.

Sie können der Methode *form_for* einen eigenen *FormBuilder* übergeben, um auf die Erzeugung des Formulars Einfluss zu nehmen, z.B. jedes Feld automatisch mit einem Label zu versehen. Ein Beispiel findet sich in der Online-Dokumentation.

Im Zuge der REST-Ausrichtung von Rails (vgl. Kapitel 7) kann der Aufruf von *form_for* weiter vereinfacht werden:

```
# projects/new.html.erb
<% form_for @project do |f| %>
  <%= f.text_field :name %> ...
<% end %>
=>
<form action="/projects" class="new_project"
      id="new_project" method="post">
```

Rails erkennt, dass die Instanz neu ist und erzeugt automatisch die Action *projects* und die HTTP-Methode *POST*. Für den *edit*-View ist der Aufruf identisch:

```
# projects/edit.html.erb
<% form_for @project do |f| %>
  <%= f.text_field :name %>
  ...
<% end %>
=>
<form action="/projects/2" class="edit_project"
      id="edit_project_2" method="post">
  <input name="_method" type="hidden" value="put" />
  ...
</form>
```

Rails erkennt, dass es sich um eine existierende Instanz handelt, und erzeugt ein Formular mit der Action `/projects/2` und dem Hidden-Field `_method`, indem es die HTTP-Methode `PUT` übergibt.

Im Zusammenhang mit `form_for` sollten Sie auch einen Blick auf die Methode `fields_for` werfen. Die Methode ermöglicht die Verarbeitung assoziierter Modelle in einem View, z.B. die Tasks zu einem Projekt.

Alternativ zu `form_for` können Sie Formulare auch mit Hilfe von `form_tag` erstellen, z.B. wenn Sie keine Modelle bearbeiten.

6.3.2 Schutz vor Cross-Site Request Forgery

Durch Cross-Site Request Forgery, kurz CSRF³ versucht ein unberechtigter Benutzer die Daten einer Webanwendung zu ändern. Er bedient sich dabei eines berechtigten Benutzers, der angemeldet ist, und schiebt diesem z.B. per E-Mail einen URL-Aufruf unter.

Rails unterstützt Sie bei der Verhinderung von CSRF, indem es in jedes Formular über das Hidden-Field⁴ `authenticity_token` automatisch einen Wert einfügt. Dieser wird beim Abschicken des Formulars mitgesendet:

```
<form action="/projects/2" method="post" ...>
  <input name="authenticity_token" type="hidden"
    value="01b8c404129b1023561637d7702d3aa1f00a0ce8" />
  ...
```

Im Controller prüft Rails automatisch per Filter `verify_authenticity_token`, ob der übertragene dem ausgelieferten Wert entspricht. Ist dies nicht der Fall, wird die Ausnahme `ActionController::InvalidAuthenticityToken` geworfen und der Aufruf der Action unterbunden.

Sofern Sie Cookies als Session-Speicher verwenden, ist dieses Verhalten automatisch aktiviert. Verwenden Sie einen anderen Session-Speicher, müssen Sie im `ApplicationController` einen eigenen Wert für die Verschlüsselung des Tokens definieren. Rails erzeugt dann aus der Session-ID und dem Wert von `secret` das Token für die Form:

Listing 6.5: `app/controllers/application.rb`

```
class ApplicationController < ActionController::Base
  # See ActionController::RequestForgeryProtection for details
  # Uncomment the :secret if you're not using
  # the cookie session store
  protect_from_forgery :secret => '1e950e65da5...'
  ...
```

Der Sicherheit halber sollten Sie keine persistenten Session-Cookies erlauben, sondern diese zeitlich begrenzen. Löschen Sie zusätzlich die Session nach einem Login, und erzeugen Sie stets eine neue. Dann sind Sie auf der sicheren Seite.

³ <http://de.wikipedia.org/wiki/Cross-Site.Request.Forgery>

⁴ vgl. Abschnitt 6.3.3

Zusätzlich sollten Sie sicherstellen, dass alle GET-Requests an Ihre Anwendung idempotent sind, d.h. keine Änderungen an den Daten vornehmen, egal wie häufig sie aufgerufen werden (vgl. Abschnitt 5.4).

6.3.3 Formular-Helper mit Bezug zu Modellen

Das Modul *FormHelper* definiert Methoden zur Erzeugung von Formularelementen, wie z.B. Eingabefelder. Die hierdurch an den Controller übermittelten Werte haben immer einen Bezug zu den Attributen eines Modells. Im folgenden Beispiel wird durch die Verwendung von *project* und *name* der Bezug zum Attribut *name* des Modells *Project* hergestellt:

```
<%= text_field :project, :name %>

=>
  <input type="text" id="project_name"
    name="project[name]" value="Rails" />
```

Das im Beispiel erzeugte Textfeld erhält den Elementnamen *project[name]* und die CSS⁵-Id *project_name*. Aufgrund des verwendeten Modellnamens *:project* erwartet *text_field*, dass eine Instanzvariable mit gleichem Namen (*@project*) in der Controller Action definiert wurde. Sie enthält die aktuell zu bearbeitende Projektinstanz, und das Textfeld wird mit dem Wert ihres Attributs *name* aus der Datenbank vorbelegt (im Beispiel mit „Rails“).

Heißt die Instanzvariable anders, so wird das Feld nicht vorbelegt. In diesem Fall müssen Sie die Instanz explizit über *:object* übergeben:

```
<%= text_field :project, :name, :object => @cms_project %>
```

Bei der Übertragung des Wertes an den Controller wird aufgrund des Parameternamens *project[name]* die Zuordnung zum Attribut *name* des Modells *Project* durch Rails hergestellt (vgl. Abschnitt 5.2.2). Jedes durch eine Methode aus *FormHelper* erzeugte HTML-Element korrespondiert auf diese Weise mit einem Attribut des Modells.

Der übergebene Projektname aus dem obigen Listing wird im Controller wie folgt aus den Request-Parametern ermittelt:

```
class ProjectsController < ApplicationController
  def create
    @project = Project.new
    @project.name = params[:project][:name]
    ...
  end
end
```

Die übertragenen Formularwerte können und werden in der Regel aber komplett an das Modell übergeben:

⁵ Cascading Style Sheets, siehe http://de.wikipedia.org/wiki/Cascading_Style_Sheets

```
class ProjectsController < ApplicationController
  def create
    @project = Project.new params[:project]
    ...
  end
end
```

Neben dem Textfeld stehen im Modul *FormHelper* weitere Methoden für Textbereiche, Checkboxes, Radiobuttons, Passwort, Dateiauswahl usw. bereit. Allen können Attribute, wie z.B. eine CSS-Klasse (per *:class*), übergeben werden. Eine Deaktivierung des Elements ist durch die Angabe von *:disabled => true* bzw. *:readonly => true* möglich.

Im Folgenden noch einige Beispiele. Wir haben der Einfachheit halber den HTML-Code für Labels usw. weggelassen:

Textbereich

Ein Textbereich wird mit Hilfe der Methode *text_area* eingefügt. Über die Parameter *cols* und *rows* können Breite und Höhe bestimmt werden:

```
<%= text_area :project, :description, :cols => 20, :rows => 5 %>
=>
  <textarea id="project_description"
    name="project[description]" cols="20" rows="5">
    ...
  </textarea>
```

Checkbox

Die Checkbox wird über die Methode *check_box* erstellt. Das zugehörige Modellattribut muss eine Ganzzahl liefern. Ist der Wert größer 0, wird die Checkbox als selektiert betrachtet, andernfalls als nicht selektiert. Da nicht selektierte Checkboxes keinen Eintrag in den Request zum Server einfügen, erzeugt Rails zusätzlich ein Hidden-Field mit demselben Namen. Dadurch wird auch bei einer nicht selektierten Checkbox ein zugehöriger Request-Parameter an den Server gesendet:

```
<%= check_box :project, :reviewed %>
=>
  <input id="project_reviewed" name="project[reviewed]"
    type="checkbox" value="1" />
  <input name="project[reviewed]" type="hidden" value="0" />
```

Radiobutton

Radiobuttons werden über die Methode *radio_button* definiert.

```
<%= radio_button :project, :status, tag_value = "green" %>
<%= radio_button :project, :status, tag_value = "red" %>
=>
  <input id="project_status_green" name="project[status]"
```

```

        type="radio" value="green" />
<input id="project_status_red" name="project[status]"
        type="radio" value="red" />

```

Je nach ausgewähltem Knopf wird in *project[status]* der Wert „green“ oder „red“ übertragen.

Versteckte Felder

Zur Übergabe von versteckten Feldern eines Formulars, so genannten *hidden fields*, steht die Methode *hidden_field* zur Verfügung:

```
<%= hidden_field :project, :last_edit %>
```

6.3.4 Formular-Helper ohne Bezug zu Modellen

Nicht jeder durch ein Formular an den Controller übertragene Wert dient der Aktualisierung eines Modulattributs. Action View stellt im Modul *FormTagHelper* Hilfsmethoden für Formularelemente ohne Modellbezug zur Verfügung.

Die Methoden haben den gleichen Namen wie die Methoden des Moduls *FormHelper*, tragen aber zusätzlich das Suffix *.tag*. Da die Methoden keinen Bezug zu einem Modell haben, sind weder das Modell noch das Attribut anzugeben. Stattdessen wird im ersten Parameter der Name des Formularelements definiert. Dieser Name dient dann als Schlüssel in der *params*-Hash, unter dem der Wert des Formularelements an den Server übertragen wird.

text_field_tag

Erzeugt ein Textfeld aus Namen und Wert:

```

<%= text_field_tag :code, "foo42", :size=>10 %>
=>
  <input id="code" name="code" size="10" type="text"
        value="foo42" />

```

password_field_tag

Ein Textfeld analog *text_field_tag*, aber vom Typ *password*. Eingegebene Zeichen werden durch einen Stern unkenntlich gemacht.

```

<%= password_field_tag :password %>
=>
  <input id="password" name="password" type="password" />

```

radio_button_tag

Die Methode erzeugt einen Radio-Button mit Namen, Wert und Status (gewählt, nicht gewählt):

```
<%= radio_button_tag :sex, "Frau, false %>
<%= radio_button_tag :sex, "Mann, true %>
=>
  <input id="sex" name="sex" type="radio" value="Frau" />
  <input checked="checked" id="sex" name="sex"
    type="radio" value="Mann" />
```

select_tag

Die Methode erzeugt das *select*-Tag, das die *option*-Tags in Auswahlboxen umschließt. Die Optionen werden z.B. über *options_for_select* eingefügt (vgl. Abschnitt 6.3.8):

```
<%= select_tag("project",
  options_for_select([ ["Java", 1], ["Rails", 2] ]),
  { :class => "select" }) %>
=>
  <select class="select" id="project" name="project">
    <option value="1">Java</option>
    <option value="2">Rails</option>
  </select>
```

6.3.5 HTML-Tags

Die Methoden *content_tag* und *tag* aus dem Modul *TagHelper* unterstützen das Schreiben von HTML-Tags. Statt selbst öffnende und schließende Tags zu codieren, übernehmen dies die Methoden. Bei *content_tag* ist auch eine Verschachtelung möglich:

```
<%= content_tag("div",
  content_tag("p", "Rails", :id=>"rails"),
  :class => "notice" ) %>
=>
  <div class="notice"><p id="rails">Rails</p></div>

<%= tag("br", { :id => "rails" } ) %>
<%= tag("br", { :id => "rails"}, true ) %>
=>
  <br id="rails" />
  <br id="rails">
```

6.3.6 Texte und Zahlen

In den Modulen *TextHelper* und *NumberHelper* finden sich Methoden zur Formatierung der Ausgaben.

Zahlen

Formatierung mit Tausendertrennzeichen:

```
<%= number_with_delimiter(123456789) %>
=> 123,456,789
<%= number_with_delimiter(123456789, '.') %>
=> 123.456.789
```

Speichergrößen in lesbarer Form:

```
<%= human_size(123) %>           => 123 Bytes
<%= human_size(1234) %>          => 1.2 KB
<%= human_size(12345) %>         => 12.1 KB
<%= human_size(1234567) %>       => 1.2 MB
<%= human_size(1234567890) %>    => 1.1 GB
```

Einige der Methoden sind für die Internationalisierung der Anwendung nicht geeignet. Wir zeigen Ihnen Lösungsmöglichkeiten in Kapitel 8.

Texte

Abschneiden von Texten durch Angabe der maximalen Textlänge:

```
<%= truncate("Ich schwimme wie ein Fisch!", 12) %>
=> "Ich schwimme"
```

Durch Einsatz der Methoden *auto_link* und *strip_links* kann einfacher Text in Links bzw. Links in einfachen Text konvertiert werden:

```
<%= auto_link("Visit http://www.b-simple.de") %>
=>
  Visit <a href="http://www.b-simple.de">
    http://www.b-simple.de</a>

<%= strip_links(' <a href="http://www.b-simple.de">
  b-simple</a>') %>
=>
  b-simple
```

6.3.7 Datum und Zeit

Das Modul *DateHelper* stellt Auswahlboxen für Datum und Zeit zur Verfügung. Die Methode *date_select* z. B. erzeugt eine Kombination aus Auswahlboxen für Tag, Monat und Jahr. Entsprechende Optionen regeln die Reihenfolge der Anzeige oder die Verwendung von Monatsnamen bzw. -zahlen. Die Auswahl eines Datums ist mit den durch die beiden ersten Parameter definierten Modellattributen assoziiert (vgl. Abschnitt 6.3.3).

```
<%= date_select("project", "started_on",
  :start_year => 2005, :order => [:day, :month, :year],
  :use_month_numbers => true, :include_blank => true) %>
=>
  <select id="project_started_on_3i" name="project[started_on(3i)]">
    ...
```

```
<select id="project_started_on_2i" name="project[started_on(2i)]">
...
<select id="project_started_on_2i" name="project[started_on(1i)]">
...
```

Da das Datum über drei Auswahlboxen selektiert wird, werden auch drei Werte an den Controller übermittelt. Damit Rails daraus ein Datum erzeugen kann, erhalten die Parameter den gleichen Namen und zusätzlich einen Index. Die Parameter für Tag, Monat und Jahr lauten für das obige Beispiel daher *project[started_on(3i)]*, *...(2i)]* und *...(1i)]*. Das *i* gibt darüber hinaus den Typ *Integer* des Wertes an.

Für die Konvertierung der drei Parameter in ein Objekt vom Typ *Date* sorgt Rails selbst, und zwar bei der Zuweisung der Parameter an das Modell:

```
class ProjectsController < ApplicationController
  def create
    @project = Project.new params[:project]
    date = @project.started_on # liefert Typ Date
    ...
  end
end
```

Über die Methode *select.date* wird die gleiche Auswahlbox auch ohne Assoziation zu einem Modellattribut bereitgestellt. Für einzelne Elemente wie Tag, Monat, Stunde, Minute usw. steht ebenso jeweils eine entsprechende Methode *select_XXX* zur Verfügung.

Datumsvalidierung

Die Eingabe eines ungültigen Datums (z.B. 31. Juni 2005) wird nicht verhindert. Dies führt im obigen Beispiel zu einer *ActiveRecord::MultiparameterAssignmentErrors*, auf die Sie entsprechend reagieren müssen.

Internationalisierung

Auch die Methoden für die Datumseingabe sind per Default nicht für die Internationalisierung der Anwendung geeignet, da Sie die Wochen- und Monatsnamen in Englisch anzeigen. In Abschnitt 8.5.10 zeigen wir Ihnen, wie Sie die Methoden für eine deutschsprachige Anwendung anpassen können.

6.3.8 Auswahlboxen

Mit Hilfe des Moduls *FormOptionHelper* werden Auswahlboxen oder Optionen für Auswahlboxen aus Arrays oder Hashes erzeugt. Ebenso existiert eine Unterstützung für die Auswahl von Ländern und Zeitzonen. Im Folgenden stellen wir die wichtigsten dieser Hilfsmethoden vor.

select(object, method, choices, options = {}, html_options = {})

Die Methode *select* erzeugt eine Auswahlbox. Diese ist mit einem Modell und Modellattribut assoziiert, dessen aktueller Wert vorselektiert ist. Über weitere Optionen kann z.B. ein Leerfeld der Auswahl hinzugefügt oder über die HTML-Optionen eine CSS-Klasse eingefügt werden.

```
<%= select("project", "name",
  [ ["Python", 1], ["Ruby", 2], ["Java", 3] ],
  { :include_blank => true },
  { :class => "projectstyle" } ) %>

=>
<select id="project_name" name="project[name]"
  class="projectstyle">
  <option value=""></option>
  <option value="1" selected="selected">Python</option>
  <option value="2">Ruby</option>
  <option value="3">Java</option>
</select>
```

options_for_select(container, selected = nil)

Die Methode *options_for_select* erzeugt für jedes Element aus einem Array einen *option*-Eintrag. Der vorselektierte Eintrag ist optional im zweiten Parameter anzugeben. Entsprechende *select*-Tags sind explizit und z.B. über *select_tag* ins HTML einzufügen:

```
<%= select_tag("project",
  options_for_select(["CMS", "CRM"], "CRM") ) %>

=>
<select id="project" name="project">
  <option value="CMS">CMS</option>
  <option value="CRM" selected="selected">CRM</option>
</select>
```

Besteht jedes Element im Array aus einem Paar, so wird der erste Wert des Paares für den Optionstext und der zweite als Optionswert für den Request verwendet:

```
<%= select_tag("project",
  options_for_select(["CMS", 1], ["CRM", 2]),
  "CRM") ) %>

</select>

=>
<select id="project" name="project">
  <option value="1">CMS</option>
  <option value="2" selected="selected">CRM</option>
</select>
```

options_from_collection_for_select(collection, value, text, selected = nil)

Die Methode erzeugt einen *option*-Eintrag für jedes Element aus einem Array. Dabei werden der Optionstext und Optionswert eines Elements über die angegebenen Methoden aus dem Element ermittelt. Optional kann ein Element vorselektiert werden.

Stellen Sie sich ein Array von Objekten des Typs *Project* vor. Jedes Projekt hat einen Namen, der über die Methode *name* ermittelt und als Option-Text verwendet wird. Der Primärschlüssel ID des Projekts wird als Option-Wert genutzt. Das Projekt mit der ID 2 ist vorselektiert.

```
<%= select_tag(:project,
  options_from_collection_for_select(
    @projects, :id, :name, 2)) %>
=>
<select id="project" name="project">
  <option value="1">CMS</option>
  <option value="2" selected="selected">CRM</option>
  ...
```

6.3.9 Verweise und URLs

Das Modul *UrlHelper* stellt Methoden zur einfachen Erzeugung von Links zur Verfügung. Die Parameter, wie z.B. *action*, entsprechen denen, die auch im Controller verwendet werden (vgl. Kapitel 5).

url_for(options = {})

Die Methode entspricht *url_for()* aus dem Modul *ActionController::Base* und ist in Abschnitt 5.12 beschrieben.

link_to(name, options = {}, html_options = nil)

Die Methode erzeugt aus dem übergebenen Namen und optionalen Parametern einen Link und verwendet dazu intern die Methode *url_for*:

```
<%= link_to "Show", { :controller => "tasks",
  :action => "show", :id => @task } %>
=>
<a href="/tasks/show/42">Show</a>
```

Analog zur Methode *url_for* kann die Angabe des Controllers entfallen. Dieser wird dann anhand des aktuellen Pfads des Views bestimmt (z.B. *tasks/show => tasks*):

```
<%= link_to "Show", { :action => "show", :id => project.id } %>
=>
<a href="/projects/show/42">Show</a>
```

Damit die komplette URL und nicht nur der relative Pfad eingefügt werden, geben Sie die Option *:only_path* an:

```
<%= link_to "List", { :action => "list", :only_path => false } %>
```



```
=>
<a href="http://localhost:3000/projects/list">List</a>
```

Entsprechende HTML-Attribute werden durch Angabe einer weiteren Hash in den Link aufgenommen:

```
<%= link_to "List", {:action => "list"}, {:class => "project"} %>
=>
<a class="project" href="/projects/list">List</a>
```

Hierdurch kann die Methode über *confirm* auch einen Bestätigungsdialog per JavaScript anzeigen:

```
<%= link_to "Löschen",
  { :action => "delete", :id => project },
  { :confirm => "Sind Sie sicher?" } %>
```

Statt der Angabe von Controller, Action usw. können und sollten Named Routes (vgl. Abschnitt 7.10) verwendet werden:

```
<%= link_to "Show", project_path(@project) %>
=>
<a href="/projects/show/42">Show</a>
```

Nutzen Sie die REST-Features von Rails (vgl. Kapitel 7), lässt sich der Aufruf für den *show*-View weiter vereinfachen, indem Sie nur die Instanz angeben:

```
<%= link_to "Show", @project %>
=>
<a href="/projects/show/42">Show</a>
```

button_to(name, options = {}, html.options = nil)

Die Methode erzeugt einen in einem Formular enthaltenen Button. Im Gegensatz zur Methode *link_to* führt der Button einen POST-Request an die angegebene Action aus. Der Einsatz wird für Situationen empfohlen, in denen ein *GET*-Request nicht sicher ist, weil er z.B. Daten aktualisiert (vgl. Kapitel 5.4).

```
<%= button_to "Aktualisieren",
  { :action => "update", :id => project }
  { :class => "button" } %>
=>
<form method="post" action="/projects/update/42"
  class="button-to">
  <div><input value="Aktualisieren" type="submit"
    class="button"/></div>
</form>
```

Die Parameter entsprechen denen von *link.to*. Wie im Beispiel zu sehen ist, wird die angegebene HTML-Option *class* dem *input*-Element zugeordnet. Über die HTML-Option *disabled => true* kann der Button deaktiviert werden.

mail_to(email_address, name = nil, html_options = {})

Mit der Methode *mail_to* wird ein Link für die Versendung einer E-Mail erzeugt. Der erste Parameter definiert die E-Mail-Adresse und der zweite den Namen des Links. Ist dieser *nil*, wird die E-Mail-Adresse verwendet. Über HTML-Optionen können die Mail-Elemente *cc*, *bcc*, *subject* und *body* angegeben werden:

```
<%= mail_to "info@b-simple.de", nil,
  :subject => "Rails Workshop" %>
=>
  <a href="mailto:info@b-simple.de?subject=
    Rails%20Workshop">Rails Workshop</a>
```

Durch die Angabe einer Verschlüsselungsart kann die Mail-Adresse vor Spidern versteckt werden:

```
<%= mail_to "info@b-simple.de", "b-simple",
  :encode => "hex" %>
=>
  <a href="mailto:%69%6e%66%6f%62-%73%69%6d%70%6c%65 \
    .%64%65">b-simple</a>
```

6.3.10 Ressourcen einbinden

Über das Modul *AssetTagHelper* stehen Methoden für das Einbinden von Stylesheets, JavaScript und Images zur Verfügung. Die Methoden sind in zweifacher Ausführung vorhanden. Alle Methoden mit der Endung *_tag* liefern einen HTML-Link zurück. Die Methoden ohne Endung liefern den Pfad zur Ressource als String ohne HTML-Code.

stylesheet_link_tag(*sources)

Die Methode liefert einen CSS-Link mit dem angegebenen Namen:

```
<%= stylesheet_link_tag "mystyle" %>
=>
  <link href="/stylesheets/mystyle.css" media="screen"
    rel="Stylesheet" type="text/css" />
```

javascript_include_tag(*sources)

Analog dem Einbinden von Stylesheets werden mit dieser Methode eine oder mehrere JavaScript-Dateien eingefügt; bei der Angabe der Option *defaults* sogar alle für Ajax benötigten Dateien plus der Datei *public/javascripts/application.js* auf einmal (vgl. Kapitel 10):

```
<%= javascript_include_tag "myscript" %>
=>
  <script src="/javascripts/myscript.js"
    type="text/javascript"></script>
```

image_tag(source, options = {})

Das Einbinden eines Images erfolgt über die Angabe des Namens und optional des Alternativtexts und der Größe. Wird kein Pfad angegeben, verwendet die Methode das Präfix */images*, wodurch das Image im Standardverzeichnis der Rails-Anwendung (*public/images*) gesucht wird.

```
<%= image_tag("buy.gif",
  { :alt => "Jetzt kaufen!", :size => "34x43" }) %>
=>
  
```

6.3.11 JavaScript

Die Methoden im Modul *JavaScriptHelper* dienen hauptsächlich zur Unterstützung von Ajax-Funktionalität und werden in Kapitel 10 beschrieben. Für das Einbinden von JavaScript-Dateien stehen ebenfalls Hilfsmethoden zur Verfügung (siehe Abschnitt 6.3.10).

6.3.12 Code speichern und wiederverwenden

Die Methoden im Modul *CaptureHelper* ermöglichen es, Code in einer Variable zu speichern und diese überall in den Templates wieder zu verwenden. So ist es z.B. mit Hilfe der Methode *content_for* möglich, in einem View JavaScript zu definieren, das im Kopfbereich des Layout Views (vgl. Abschnitt 6.4) eingefügt wird.

```
# project.html.erb
<% content_for :script do %>
  alert('Hallo Rails!')
<% end %>

# layout.html.erb
<head>
  <script type="text/javascript">
    <%= yield :script %>
  </script>
</head>
...
```

Mit Hilfe der Methode *capture* besteht die Möglichkeit, einen ganzen View in einer Variable zu speichern und ihn so ggf. vor dem Zurückliefern zu filtern.

```
# project.html.erb
<% page = capture do %>
  ...view code...
<% end %>

<%= filter_page page %>
```

6.3.13 Debugging

Während der Entwicklung kann es notwendig sein, die Inhalte von Arrays oder Hashes zu Debugging-Zwecken in der Seite auszugeben. Dafür steht die Methode *debug* zur Verfügung, die den Inhalt lesbar formatiert ausgibt.

```
<%= debug(project) %>

=>
  --- !ruby/object:Project
  attributes:
    name: Mein erstes Rails-Projekt
    id: "1"
    description: ...
    lead: Thomas
```

6.3.14 HTML-Code filtern

Das durch den Ruby-Code eingefügte HTML kann unter Umständen selbst HTML oder JavaScript enthalten. Dies ist z.B. bei Kommentaren in Blogs häufig der Fall. Auf diese Weise wird das so genannte Cross-Site Scripting⁶ ermöglicht (vgl. Abschnitt 11.6).

Die folgenden Helper filtern den Code, bevor er in den View gelangt. Sie sollten diese Methoden immer verwenden, damit die Anwendung in diesem Bereich sicher ist.

html.escape

Die Methode *html.escape* bzw. die Kurzform *h* filtern jeden HTML-Code aus dem String

```
<%=h "Mein Name ist <script>alert('Gnep');</script>" %>

=>
  Mein Name ist &lt;script&gt;alert('Gnep');&lt;/script&gt;
```

Um nicht jedesmal an die Nutzung der Methode zu denken, können Sie alternativ das Plugin *SafeERB*⁷ einsetzen. Gehen Sie auf Nummer sicher!

sanitize

Möchten Sie in Ihrer Anwendung bestimmte Formatierungen über HTML zulassen und andere ausschließen, so verwenden Sie die Methode *sanitize*. Diese filtert alle HTML-Tags aus dem Code, die nicht explizit erlaubt sind, u.a. auch *href*- und *src*-Attribute, die *javascript*: enthalten:

```
<%=sanitize "Mein Name ist <script>alert('Gnep');</script>" %>
<%=sanitize "Mein Name ist <b>Gnep</b>" %>

=>
```

⁶ http://de.wikipedia.org/wiki/Cross-Site_Scripting

⁷ http://agilewebdevelopment.com/plugins/safe_erb

```
Mein Name ist  
Mein Name ist <b>Gnep</b>
```

Sie können die erlaubten Tags konfigurieren. Wir verweisen hierzu auf die Online-Dokumentation. Alternativ steht Ihnen das Whitelist-Plugin⁸ zur Verfügung.

6.4 Layouts

Um ein Grundlayout der Seiten (z.B. Kopf, Inhaltsbereich und Fußzeile) nicht in jedem View zu duplizieren, bietet Rails die Möglichkeit, Layouts zentral zu definieren. Ein Layout ist ein „normales“ RHTML-Template, das die Ruby-Anweisung *yield* enthält:

```
<html>  
  <!-- header -->  
  ...  
  <!-- content -->  
  <%= yield %>  
  ...  
  <!-- footer -->  
  ...  
</html>
```

Ist ein solches Layout als Template definiert, wird das von der Action zurückgelieferte HTML nicht direkt an den Browser ausgeliefert. Stattdessen wird es innerhalb des Layouts an die Stelle der *yield*-Anweisung eingefügt und der so entstehende HTML-Code zurückgesandt.

Alle Layouts werden im Verzeichnis *app/views/layouts* relativ zum Hauptverzeichnis des Projekts abgelegt. Ein für die gesamte Anwendung gültiges Layout kann durch ein Template mit dem Namen *application.html.erb* definiert werden. Findet Rails ein solches Template, wird das enthaltene Layout automatisch bei jeder Action eines jeden Controllers verwendet.

Controller-spezifische Layouts werden ermöglicht, indem die Datei den Namen des Controllers trägt. Existiert für einen Controller namens *Projects* ein Template *projects.html.erb*, wird für alle Actions dieses Controllers nicht mehr das Layout aus *application.html.erb*, sondern jenes aus *projects.html.erb* verwendet. Das für einen Controller zu verwendende Layout kann auch explizit im Controller über die Methode *layout* definiert werden (vgl. Abschnitt 5.13).

Dem Layout eines Controllers stehen die gleichen Instanzvariablen zur Verfügung, die auch für das eigentliche Template der Action bereitgestellt werden.

6.4.1 Layout aus Views beeinflussen

Mit Hilfe der Methode *content_for* können Sie aus einem View auch Einfluss auf das Layout nehmen. Um z.B. den Seitentitel eines Views im Layout anzuzeigen, ihn aber

⁸ <http://svn.techno-weenie.net/projects/plugins/white.list>

im View selbst zu definieren, gehen Sie wie folgt vor. Der View selbst definiert den Titel inklusive HTML-Code unter dem Schlüssel *:page_title*:

```
# projects/edit.html.erb
<% content_for :page_title do %>
  <h2>Edit Project</h2>
<% end %>
... Edit-Formular...
```

Die Anweisungen zu *content_for* erscheinen nicht im View. Stattdessen fügen Sie den Code aus dem Block über den Aufruf *yield :page_title* in das Layout ein:

```
# app/views/layouts/application.html.erb
<h1>AddressBook</h1>
<%= yield :page_title %>
...
<%= yield %>
...
```

Auf diese Weise können Sie auch die Sidebar, Navigation usw. aus den Views heraus beeinflussen.

6.5 Partial Views

Häufig wird ein Teilbereich eines Views auch für andere Views benötigt. Es ist z.B. denkbar, das HTML für die Anzeige eines Projekt-Tasks in einen eigenen Teil-View auszulagern. Dieser wird überall dort eingebunden, wo Tasks anzuzeigen sind.⁹ Dies ist ein Beispiel für die Einhaltung des DRY-Prinzips, da der HTML-Code so nicht in vielen Templates dupliziert wird.

Dieses Konzept wird bei Rails über die so genannten *Partial Views* realisiert. Dabei handelt es sich um gewöhnliche Templates, denen Daten aus dem aufrufenden Template übergeben werden.¹⁰

Der Dateiname eines Partial View Templates beginnt per Konvention immer mit einem Unterstrich. Dieser dient der Unterscheidung gegenüber den Haupt-Templates, wird aber bei der Verwendung des Template-Namens weggelassen. Im folgenden Beispiel wird aus einem zum *ProjectsController* gehörenden View *index.html.erb* der Partial View *project* aufgerufen. Der Controller sucht daher das Template *app/view/projects/_project.html.erb*.

```
# projects/index.html.erb
<%= render :partial => "project" %>

# projects/_project.html.erb
<%= project.name %>
...
```

⁹ Ähnlich der Verlagerung von Code in eine Methode, die überall aufgerufen werden kann.

¹⁰ Ähnlich der Übergabe von Argumenten an eine Methode.

Der Aufruf des Partial Views erfolgt durch die Methode *render*. Dieser wird mindestens der Parameter *partial* mit dem Namen des Partial Views übergeben. Ohne weitere Parameter wird implizit der Wert aus der Instanzvariable *@project* übergeben. Diese ist vorher entsprechend in der Controller Action zu definieren (vgl. Abschnitt 5.2.1).

Im Partial View steht der Wert über die lokale Variable *project* zur Verfügung. Da Rails die Variablen aus dem Namen des Templates herleitet, muss der Name des Partial Views neben einem gültigen Dateinamen auch ein gültiger Ruby-Variablenname sein.

Partial Views und Parameter

Alternativ kann der Wert auch explizit über den Parameter *:object* übergeben werden, wobei der Parameternamen vom Template-Namen abweichen kann. Im Partial View steht der Wert nach wie vor über die lokale Variable *project* zur Verfügung:

```
# projects/index.html.erb
<%= render :partial => "project", :object => @my_project %>

# projects/_project.html.erb
<%= project.name %>
...
```

Der Aufruf lässt sich weiter vereinfachen, indem die Instanzvariable direkt angegeben wird:

```
# projects/index.html.erb
<%= render :partial => @my_project %>
...
```

Rails leitet den Namen des Partial Views aus der Klasse der Instanzvariable ab (hier *Project*) und sucht daher nach der Datei *_project.html.erb* unter dem Verzeichnis *app/views/projects/*¹¹:

```
# projects/_project.html.erb
<%= project.name %>
```

Über die Hash *locals* können dem Partial View weitere Argumente übergeben werden. Durch die Schlüsselnamen sind zugleich die Namen der lokalen Variablen im Partial View festgelegt. Im folgenden Beispiel werden die Werte aus *@project* (implizit) und *@project* (explizit) übergeben und stehen im View über die lokalen Variablen *project* und *prj* zur Verfügung:

```
# projects/index.html.erb
<%= render :partial => "project",
  :locals => { :prj => @project } %>

# projects/_project.html.erb
<h4>Status of Project: <%= prj.name %></h4>
```

¹¹ Beachten Sie den Plural projects, nicht project.

```
<%= project.count_tasks %>
...
```

Innerhalb des Partial View sind alle lokalen Variablen in der Hash *local_assigns* enthalten. Sie können darüber z.B. die Existenz einer lokalen Variable prüfen:

```
# projects/_project.html.erb %>
<% if local_assigns.include?(:name) %>
  Name: <%= name %>
<% end %>

# projects/_project.html.erb %>
<% name = local_assigns[:name] || 'NoName'
Name: <%= name %>
...
```

Partial Views und Instanzvariablen

Jede in der Controller Action definierte Instanzvariable steht auch in jedem im Template eingebundenen Partial View zur Verfügung. Es ist im Beispiel also auch folgender Aufruf möglich:

```
# projects/index.html.erb
<%= render :partial => "project" %>

# projects/_project.html.erb
<%= @project.name %>
```

Von der Verwendung von Instanzvariablen in Partial Views raten wir Ihnen aber ab. Sie müssen nämlich sonst den Überblick behalten, von wo der Partial View überall eingebunden wird und ob die entsprechende Controller Action die Instanzvariablen denn auch definiert.

Partial Views iterativ verwenden

Mit Hilfe der Methode *render* kann für jedes Objekt aus einem Array der Partial View iterativ verwendet werden. Hierzu ist das Array über den Parameter *:collection* an die Methode zu übergeben. Die Methode *render* sorgt dann automatisch für die Iteration über das Array und den Aufruf des Partial Views pro Element.

In jedem View steht Ihnen der Array-Index über die lokale Variable *<Partial View>_counter* zur Verfügung. Zur visuellen Trennung der einzelnen Views kann über den Parameter *:spacer_template* ein Template angegeben werden.

Das folgende Beispiel erzeugt eine Übersichtsliste aller Projekte. Die Daten jedes einzelnen Projekts werden dabei durch den Partial View *project* inklusive laufender Nummer angezeigt. Jeder View wird durch das HTML aus dem Template *app/views/projects/_line_spacer.html.erb* optisch vom nächsten getrennt. Die Projektdaten stehen jeweils über die lokale Variable *project* bereit.

```
# projects/index.html.erb
<%= render :partial => "project",
```



```

      :collection => @projects,
      :spacer_template => "line_spacer" %>

# projects/_project.html.erb
[<%= project_counter %>] <%= project.name %>
...

```

Der Aufruf kann vereinfacht werden, indem die Instanzvariable direkt angegeben wird:

```

# projects/index.html.erb
<%= render :partial => @projects %>
...

```

Rails erkennt die Angabe eines Array von Projekten und sucht nach dem Partial View *app/views/projects/_project.html.erb*, der pro Instanz eingebunden wird.

Partial View Layouts

Die Anzeige eines Partial View kann über die Angabe eines *Partial View Layouts* variiert werden. So kann z.B. der Partial View zur Anzeige des Status mit einem einfachen Layout versehen werden, das nur die Überschrift und einen Link auf Details beinhaltet:

```

# projects/show.html.erb
<h2>Project</h2>
...
<%= render :partial => @project, :layout => 'project_simple' %>

# projects/_project.html.erb
...Name: <%= project.name %>...

# projects/_project_simple.html.erb
...Überschrift...
<%= yield %>

=>
  <h2>Project</h2>
  ...Überschrift...
  ...Name: CMS...

```

Derselbe Partial View wird über die Angabe des Layouts *project_edit* mit Links zur Bearbeitung versehen:

```

# projects/show.html.erb
<h2>Project</h2>
...
<%= render :partial => "project", :layout => 'project_edit' %>

# projects/_project.html.erb
...Name: <%= project.name %>...

```

```
# projects/_project_edit.html.erb
...Bearbeiten: ... Links ...
<%= yield %>

=>
  <h2>Project</h2>
  ...Bearbeiten: ... Links ...
  ...<%= project.name %>...
  ...Name: CMS...
```

Partial Views eines anderen Controllers verwenden

Die Partial Views eines Controllers können auch von anderen Controllern verwendet werden. Dazu ist der Name des Controllers bzw. der Name seines Verzeichnisses als Präfix zum Template-Namen anzugeben.

```
# projects/show.html.erb
<%= render :partial => "tasks/task",
  :object => @project.tasks.first %>

# tasks/_task.html.erb
<%= task.name %>, ...
...
```

6.6 Anzeige Fehlermeldungen

Für die Anzeige von Validierungsfehlern in einem Modell bietet das Modul *ActiveRecordHelper* Hilfsmethoden. Die Methode *error_messages_for* gibt alle Meldungen zu einem Modell aus (vgl. 3.11):

```
<%= error_messages_for :address %>
=>
  <div class="errorExplanation" id="errorExplanation">
    <h2>3 errors prohibited this address from being saved
    <p>There were problems with the following fields:</p>
    ...
  </div>
```

Des Weiteren steht die Methode *error_message_on* zur Verfügung, die den Modellnamen und das Attribut entgegennimmt und HTML-Code mit der Fehlermeldung liefert.

```
<%= error_message_on :address, :name %>
=>
  <div class="formError">can't be blank</div>
```

Dazu müssen die Attribute des Modells bekannt sein, die einen oder mehrere Fehler verursacht haben. Wie in Abschnitt 4.14 beschrieben, enthält jedes Modell ein Attribut *errors* mit allen Fehlermeldungen. Über entsprechende Iteratoren *each* oder *each_full* können Sie der Reihe nach an alle Meldungen gelangen.

Die Anzeige der Fehlermeldungen ist allerdings sehr einfach gehalten und für eine professionelle Anwendung wenig benutzerfreundlich. Die Reihenfolge der Meldungen ist willkürlich und passt nicht zwangsläufig zur Anordnung der Felder auf der Seite. Außerdem sind die Überschriften und Fehlermeldungen per Default in Englisch.

6.7 XML-Templates

Neben HTML unterstützt Rails auch die Auslieferung von XML, z.B. für RSS¹²-Dateien. Das XML wird in entsprechenden XML-Templates definiert, die per Default ebenfalls im Template-Verzeichnis eines Controllers liegen. Die Dateien haben die bevorzugte Dateieindung *.builder* oder die ältere *.xml*.

Der Aufruf aus dem Controller unterscheidet sich nicht von dem eines HTML-Templates. Da XML-Templates aber für die Auslieferung von XML gedacht sind, sollten diese nicht innerhalb eines HTML-Layouts verwendet werden. Deaktivieren Sie ggf. ein zum Controller vorhandenes Layout innerhalb der Action, die XML liefern soll. Oder lagern Sie die XML-Erzeugung in einen eigenen Controller ohne Layout aus.

```
class ProjectsController < ApplicationController
  def projects_as_xml
    @projects = Project.find(:all)
    render :layout => false
  end
end
```

Per Default steht in jedem XML-Template eine Instanz eines XML-Erzeugers über die Variable *xml* zur Verfügung. Auf ihr aufgerufene Methoden werden zu einem XML-Tag mit Namen der Methode konvertiert. Übergebene Parameter definieren dabei den Text und die Attribute zum Tag.

```
xml.project("Rails", :class => "pname")
=>
<project class="pname">Rails</project>
```

Kollidiert der Name eines Tags mit dem einer auf *xml* bestehenden Methode, erfolgt die Erzeugung über die Methode *tag!*.

```
xml.tag!("id", project.id)
=>
<id>42</id>
```

Auf diese Weise werden komplexere XML-Strukturen (z.B. für RSS-Feeds) elegant und lesbar geschrieben. Ist eine Liste aller Projekte als XML zu liefern, könnte dies wie im folgenden Beispiel aussehen. Kommentarzeilen werden dabei wie in Ruby durch die Raute *#* gekennzeichnet:

¹² Siehe <http://de.wikipedia.org/wiki/RSS>

```

xml.instruct! :xml, :version=>"1.0", :encoding=>"UTF-8"
# list all projects
xml.projects do
  @projects.each do |project|
    xml.project do
      xml.name(project.name, :class => "pname")
      xml.description(project.description)
    end
  end
end

=>
<projects>
  <project>
    <name class="pname">My first Ruby project</name>
    <description>The project ...</description>
  </project>
  <project>
    <name class="pname">Your second Rails project</name>
    <description>...</description>
  </project>
  ...
</projects>

```

6.8 RJS-Templates

Ein weiteres View-Format stellen die so genannten *Remote JavaScript Templates* dar, die wir in Kapitel 10 beschreiben.

6.9 Caching

Ein zentraler Mechanismus für die Entwicklung von performanten Rails-Anwendungen ist das *Caching*. Beim Caching werden dynamische Seiten nach ihrer Erzeugung durch einen initialen HTTP-Request zwischengespeichert und bei Folge-requests direkt aus dem Cache geliefert und somit nicht neu generiert. Ein Großteil der Requests einer Web-Anwendung liefert immer wieder dieselben Daten, sodass Caching zu einer erheblich besseren Performance der Anwendung führt. Denken Sie nur einmal an die in Deutschland sehr populäre und entsprechend häufig angewählte News-Seite des Heise-Verlags (<http://www.heise.de/newsticker/>), die für sehr viele aufeinander folgende Requests immer wieder denselben Inhalt liefert.

Caching ist standardmäßig nur für die Produktionsumgebung aktiviert und kann durch den folgenden Aufruf in einer der drei Umgebungs-Konfigurationsdateien in *config/environments* aktiviert bzw. deaktiviert werden:

```
config.action_controller.perform_caching = true bzw. false
```

Sollte die Definition keine Auswirkung haben, versuchen Sie es mit der alten Konfigurationseinstellung:

```
ActionController::Base.perform_caching = true bzw. false
```

Rails unterscheidet drei Caching-Varianten: Seiten-Caching, Action-Caching und Fragment-Caching. Seiten- und Action-Caching findet auf Controller-Ebene, Fragment-Caching hingegen auf View-Ebene statt.

6.9.1 Seiten-Caching

Seiten-Caching ist die einfachste Caching-Variante und wird im Controller durch Aufruf der Methode *caches_page* aktiviert. Die Methode erwartet eine oder mehrere Actions, für die das Seiten-Caching aktiviert werden soll:

```
class NewsController < ApplicationController
  caches_page :overview
end
```

Actions mit aktiviertem Seiten-Caching erzeugen die Seite nur beim ersten Aufruf. Dabei wird die Seite als HTML-Datei im Caching-Verzeichnis (*ActionController::Base.page_cache_directory*) erzeugt, das in der Produktionsumgebung auf *public/* zeigt. Für alle Folgeaufrufe liegt die Seite als statische Datei vor und wird direkt vom Webserver zurückgeliefert, d.h. die Rails-Anwendung ist in die Bearbeitung des Requests nicht mehr involviert.

6.9.2 Action-Caching

Action-Caching ist die zweite Controller-basierte Caching-Variante. Sie unterscheidet sich vom Seiten-Caching insofern, als die Before-Filter der jeweiligen Action auf jeden Fall ausgeführt werden. Nur wenn alle Before-Filter erfolgreich durchlaufen werden, wird die angeforderte Seite aus dem Cache geliefert. Action-Caching wird für einzelne Actions durch Aufruf der Methode *caches_action* aktiviert:

```
class NewsController < ApplicationController
  caches_action :overview_for_subscribed_users
  before_filter :authenticate, :except => [:login, :sign_on]

  def overview_for_subscribed_users
    ...
  end
  ...
end
```

Im obigen Beispiel stellt der News-Controller neben der allgemeinen und öffentlich zugänglichen News-Seite eine spezielle News-Übersicht für zahlende Nutzer bereit: *overview_for_subscribed_users*. Vor der Ausführung dieser Action muss geprüft werden, ob der aktuelle Benutzer beim System angemeldet und damit zum Aufruf der Seite berechtigt ist. Die Überprüfung wird von der Methode *authenticate* vorgenommen, die mit Hilfe eines Before-Filters vor Ausführung jeder Action (bis

auf `:login` und `:sign_on`) aktiviert wird. Der Aufruf `cache_action` stellt sicher, dass der Filter auch bei eingeschaltetem Caching aufgerufen wird.

Freigabe gecachter Seiten

Die Erzeugung von gecachten Seiten ist die eine Seite der Medaille. Die andere Seite ist die Beantwortung der Frage, nach welchen Regeln gecachte Seiten gelöscht werden sollen, weil sich z.B. die für die Erzeugung der Seiten verwendeten Datenbankinhalte geändert haben.

Für das explizite Löschen von Seiten aus dem Cache stellt Rails die beiden Controller-Methoden `expire_page` und `expire_action` zur Verfügung. Beide Methoden erwarten als Parameter eine Action und eine Reihe weiterer optionaler Parameter, wie z.B. einen Controller oder eine ID, sofern sich die zu löschende Seite auf eine bestimmte Modellinstanz bezieht. Die Methoden werden direkt im Quellcode an Stellen verwendet, an denen die Ungültigkeit explizit deutlich wird. Fügt der News-Administrator der beschriebenen News-Site einen weiteren Eintrag hinzu, dann ist in der entsprechenden Controller-Action explizit klar, dass die gecachte Nachrichten-Übersichtsseite ungültig ist und durch einen Aufruf von `expire_page` aus dem Cache gelöscht werden muss:

```
class NewsController < ApplicationController

  def add_message
    ...
    # Seite beim nächsten Request frisch erzeugen
    expire_page :action => 'overview'
  end
  ...
end
```

Mittels Action-Caching gecachte Seiten müssen mit `expire_action` aus dem Cache entfernt werden:

```
class NewsController < ApplicationController

  def add_message_for_subscribers
    ...
    # Seite beim nächsten Request frisch erzeugen
    expire_action :action => 'overview'
  end
  ...
end
```

Neben den Methoden zum expliziten Löschen von Seiten aus dem Cache bietet Rails mit so genannten *Sweepers* die Möglichkeit zur Entwicklung von Modell-Beobachtern. Sweeper reagieren auf Modelländerungen, indem sie Seiten aus dem Cache löschen, die auf den veränderten Modelldaten basieren. Sweeper erben von `ActionController::Caching::Sweeper` und werden im Modellverzeichnis `app/models` gespeichert. Der folgende Sweeper ist für das Löschen der Übersichtsseite im beschriebenen News-Ticker zuständig:

```
class MessageSweeper < ActionController::Caching::Sweeper
  observe Message

  def after_create
    expire_page :controller => 'news', action => 'overview'
  end
  ...
end
```

Der Sweeper beobachtet Modellobjekte der Klasse *Message* und erklärt die Übersichtsseite für ungültig, nachdem eine neue Nachricht erzeugt wurde. Sweeper sind nicht per Default aktiv, sondern müssen explizit in einer Controllerklasse aktiviert werden:

```
class NewsController < ActionController
  cache_sweeper :message_sweeper, :only => [:add_message]
end
```

Die Definition von *cache_sweeper* im *NewsControllers* aktiviert den *MessageSweeper* für die Action *add_message*.

6.9.3 Fragment-Caching

Die dritte Form des Cachings, das *Fragment-Caching*, findet direkt im View statt. Damit lassen sich statische Bereiche einer Seite cachen, indem sie mit Hilfe der Methode *cache* in einen Block eingeschlossen werden:

```
... dynamischer Inhalt ...
<% cache do %>
  ... statischer Inhalt ...
<% end %>
... dynamischer Inhalt ...
```

Im Beispiel der News-Seite enthält die Einstiegsseite für registrierte Benutzer z.B. einen Kopfbereich mit persönlichen Daten und einer persönlichen Begrüßung. Dieser Bereich ist dynamisch und kann nicht gecacht werden. Der darunter befindliche Bereich mit den Top-News des Tages ist hingegen für alle Anwender gleich und ein Kandidat für das Fragment-Caching. Der Bereich wird beim ersten Request erzeugt und bei allen weiteren aus dem Cache geholt:

Listing 6.6: news.html.erb

```
<%= @personal_data %>
<% cache do %>
  <%= render :partial => "news", :collection => @top_news %>
<% end %>
```

Unabhängig vom View wird jedoch bei jedem Request die Action auf dem News-Controller aufgerufen und ermittelt die Top-News jedes Mal erneut aus der Datenbank. Um diesen unnötigen Datenbank-Aufruf zu umgehen, wird mit Hilfe der Methode *read_fragment* geprüft, ob der Bereich im Cache vorliegt:

Listing 6.7: news.controller.rb

```
class NewsController < ActionController
  def news
    @personal_data = ...
    unless read_fragment(:action => "news")
      @top_news = get_news("top")
    end
  end
end
```

Der der Methode übergebene Action-Name *news* dient als Schlüssel im Cache und ermöglicht die Zuordnung der Action zu dem korrespondierenden gecachten Bereich. Wie wir gleich sehen, kann der Schlüssel erweitert werden.

Im Laufe des Tages gibt es neue Ereignisse, die als Top-Neuigkeit anzuzeigen sind. Der Cache mit den Top-News ist somit veraltet und zu löschen. Hierzu dient die Methode *expire_fragment*. Der Methode wird dazu der Schlüssel des aus dem Cache zu löschenden Bereichs übergeben:

Listing 6.8: news.controller.rb

```
class NewsController < ActionController
  def add_top_news
    ...
    expire_fragment(:action => "news")
    redirect_to :action => "news"
  end
end
```

Der Methode *read_fragment* bzw. *expire_fragment* kann man mehrere Werte übergeben, wodurch unterschiedliche Bereiche in einem View unterschiedlich aktualisiert werden. Dazu muss die Methode *cache* ebenfalls den Schlüssel explizit übergeben bekommen, damit der Bereich in der Seite eindeutig identifiziert wird:

```
<%= @personal_data %>
<% cache(:action => "news", :part => "top_news") do %>
  <%= render :partial => "news", :collection => @top_news %>
<% end %>
<% cache(:action => "news", :part => "sport_news") do %>
  <%= render :partial => "news", :collection => @sport_news %>
<% end %>
```

Je nachdem, welche Parameter an *expire_fragment* übergeben werden, wird der jeweilige Bereich aus dem Cache gelöscht:

```
class NewsController < ActionController
  def add_top_news
    ...
    expire_fragment(:action => "news", :part => "top_news")
    redirect_to :action => "news"
  end
end
```



```

def expire_all
  expire_fragment(:action => "news", :part => "top_news")
  expire_fragment(:action => "news", :part => "sport_news")
  redirect_to :action => "news"
end
end

```

Fehler durch parallele Requests

Im Falle des Fragment-Caching kann es in Produktion durch parallele Requests im Extremfall zu Fehlern kommen.¹³

Die Anwendung wird durch mehrere Instanzen des Webservers ausgeliefert (vgl. 11.2.2). Die erste Instanz des Webservers durchläuft den Code aus Listing 6.7 und erkennt, dass der Cache befüllt ist. Bevor sie den View rendert, invalidiert die zweite Instanz des Webservers durch einen parallelen Request an *add_top_news* den Cache. Die erste Instanz durchläuft den Views aus Listing 6.6, erkennt, dass das Fragment neu zu erzeugen ist, und führt den Code aus, d.h. rendert den Partial View. Dieser nutzt die Instanzvariable *@top_news*, die aber im Controller nicht erzeugt wurde und *nil* enthält. Es kommt zu einer Ausnahme.

Erwarten Sie sehr viele Anfragen an Ihre Anwendung, sollten Sie ggf. sicherstellen, dass die zeitlich teure Operation direkt im View aufgerufen wird. Der View bildet die zentrale Schnittstelle zwischen den Instanzen der Webserver und hier kann das Problem nicht auftreten:

```

<% cache do %>
  <% @top_news = get_news("top") %>
  <%= render :partial => "news", :collection => @top_news %>
<% end %

```

Alternativ lagern Sie den Code in eine Helper-Methode aus, die Sie im View nutzen:

```

<% cache do %>
  <%= display_top_news %>
<% end %

```

Fragment-Caching-Speicher

Die gecachten Bereiche der Seite werden in Abhängigkeit des Konfigurationsparameters *ActionController::Base.fragment_cache_store* gespeichert. Es stehen dabei folgende Optionen zur Verfügung:

ActionController::Caching::Fragments::MemoryStore.new

Die Speicherung findet im Hauptspeicher statt, was sehr schnell ist, aber nicht skaliert.

ActionController::Caching::Fragments::FileStore.new(Pfad)

Die Speicherung findet im Dateisystem statt, z.B. *FileStore.new("Pfad-zum-Cache-Verzeichnis")*.

¹³ Dank an Tammo Freese für diesen Hinweis.

ActionController::Caching::Fragments::DRbStore.new(URL)

Ein externer Distributed Ruby Server (DRb) wird für die Speicherung verwendet, z.B. *FileStore.new("druby://localhost:9192")*.¹⁴

ActionController::Caching::Fragments::FileStore.new(Rechner)

Statt eines Pfades wird ein Rechner angegeben (z.B. *FileStore.new("localhost")*), wodurch ein Danga MemCached¹⁵ Verwendung findet.

6.9.4 Was Sie nicht cachen sollten

Caching arbeitet rein URL-basiert, d.h. eine gecachte Seite wird ausschließlich auf Basis der in der URL enthaltenen Informationen geliefert. Benötigt eine Seite Informationen, die nicht in der URL enthalten sind, dann ist diese Seite fürs Caching ungeeignet.

- wenn die Seite auf Datenbankinhalten basiert, die auch von anderen Anwendungen verändert werden können, ohne dass Ihre Rails-Anwendung davon etwas mitbekommt;
- wenn die Seite Informationen aus der aktuellen HTTP-Session benötigt, wie z.B. für die Darstellung personalisierter Bereiche;
- wenn die Inhalte der Seite aufgrund von zeitlichen Restriktionen ungültig werden, wie z.B. aktuelle Aktienkurse.

6.9.5 Caching testen

Wenn Sie Caching einsetzen, sollten Sie die korrekte Funktionsweise automatisiert testen. Dazu kann Ihnen das Plugin *Rails Cache Test Plugin* unter <http://blog.cosinux.org/pages/page-cache-test> helfen.

6.9.6 Action Cache Plugin

Unter der Seite <http://agilewebdevelopment.com/plugins/action.cache> finden Sie das Plugin *action.cache*. Es stellt eine Alternative zur Version aus dem Rails-Framework dar und bietet eine Reihe von Erweiterungen. Prüfen Sie das Plugin, wenn Sie Caching einsetzen müssen.

¹⁴ Siehe auch <http://wiki.rubyonrails.com/rails/pages/HowtoSetupDistributedRuby>

¹⁵ Siehe <http://www.danga.com/memcached>

Kapitel 7

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 ohnehin 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 Kapitel geht es um die Erweiterung des Sprachrepertoires von Web-Entwicklern um die HTTP-Methoden PUT und DELETE. Die Manipulation von Ressourcen unter Verwendung der HTTP-Methoden GET, POST, PUT und DELETE wird unter dem Begriff REST subsumiert, wofür Rails seit Version 1.2 Unterstützung bietet. Mehr noch: Ab Version 2 wird REST zum Standard-Paradigma für die Entwicklung von Rails-Anwendungen.

Das Kapitel 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 Kapitel 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 Kapitel schließt mit je einem Abschnitt über REST und AJAX und das Schreiben von Tests für REST-Controller sowie einer Einführung in ActiveResource, der clientseitigen Implementierung von REST innerhalb des Rails-Frameworks.

7.1 Was ist REST?

Der Begriff REST stammt aus der Dissertation von Roy Fielding [28] 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 können in unterschiedlichen Formaten repräsentiert werden (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 traditionellen¹ Rails-Anwendungen, ein Modell und die darauf anzuwendende Aktion, sondern nur noch die Ressource selber (siehe Abbildung 7.1).

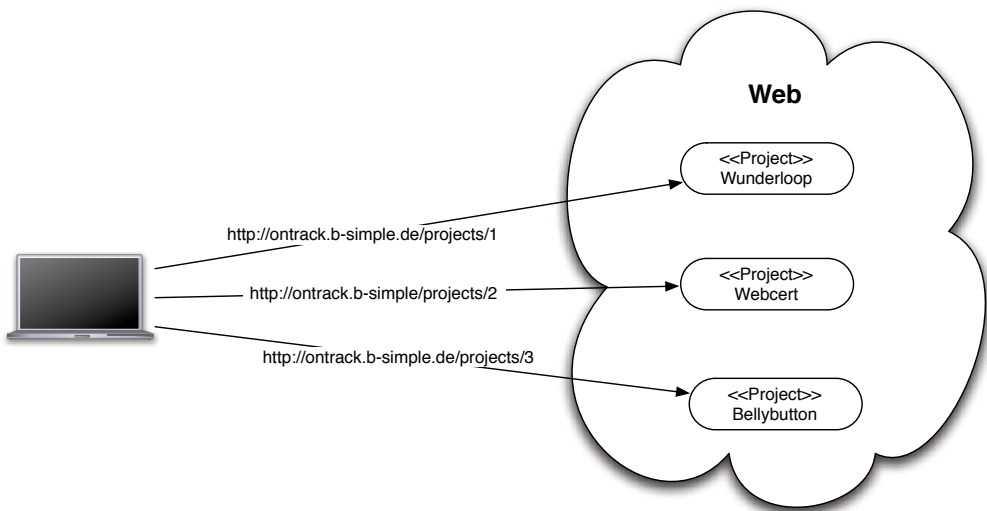


Abbildung 7.1: Ressourcen im Web und ihre URLs

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 7.1 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.

7.2 Warum REST?

Dies ist eine berechtigte Frage, insbesondere, wenn man bedenkt, dass wir seit mehr als drei 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 demselben 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 DRY² 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 einem konzeptionell sauberen und wartbaren Anwendungsdesign.

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

7.3 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

7.4 Vorbereitung

Wir beschreiben die REST-spezifischen Neuerungen in Rails am Beispiel der in Kapitel 3 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_rest
```

Es folgt die Anlage der Development- und Testdatenbank:

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

mysql> create database ontrack_rest_development;
mysql> create database ontrack_rest_test;
mysql> quit
```

7.5 Ressource Scaffolding

REST-orientierte Rails-Entwicklung lässt sich am einfachsten am Beispiel einer vom Scaffold-Generator *scaffold* 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_rest
$ ruby script/generate scaffold project name:string desc:text
  exists  app/models/
  exists  app/controllers/
  exists  app/helpers/
  create  app/views/projects
  exists  app/views/layouts/
  exists  test/functional/
  exists  test/unit/
  create  app/views/projects/index.html.erb
  create  app/views/projects/show.html.erb
  create  app/views/projects/new.html.erb
  create  app/views/projects/edit.html.erb
  create  app/views/layouts/projects.html.erb
  create  public/stylesheets/scaffold.css
dependency model
  exists  app/models/
  exists  test/unit/
  exists  test/fixtures/
  create  app/models/project.rb
  create  test/unit/project_test.rb
  create  test/fixtures/projects.yml
```

```
create    db/migrate
create    db/migrate/001_create_projects.rb
create    app/controllers/projects_controller.rb
create    test/functional/projects_controller_test.rb
create    app/helpers/projects_helper.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.

7.6 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
```

7.7 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 Action³ zur Verfügung stellt:

Listing 7.1: *ontrack_rest/app/controllers/projects_controller.rb*

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

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

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.


```
# 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 aus, d.h. wie traditionelle Controller. 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.

7.7.1 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? Eine Antwort auf diese Frage liefert die verwendete HTTP-Methode⁴, 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:

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 Erzeu-

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

Tabelle 7.1: Standard 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

gung 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 7.8.3). Gleiches gilt für PUT-Requests zur Neuanlage von Ressourcen (siehe Abschnitt 7.8.2).

7.7.2 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 7.2: `ontrack_rest/app/controllers/projects_controller.rb`

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

  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.

7.7.3 Accept-Feld im HTTP-Header

Beginnen wir mit 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) ...
[2006-12-30 18:10:50] INFO WEBrick::HTTPServer#start: pid=...
```

Erfassen Sie anschließend ein oder mehrere Projekte über einen Browser (siehe Abbildung 7.2).

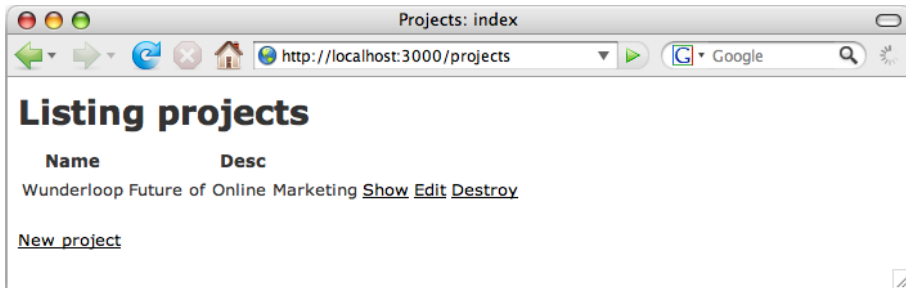


Abbildung 7.2: Initiale Projekterfassung

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=4545eabd9d1bebbe367ecbadf015bcc2; 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 *ProjectsControllers*. Beachten Sie, dass die URL identisch zum vorherigen curl-Aufruf ist. Der einzige Unterschied ist das verwendete HTTP-Verb.

7.7.4 Formatangabe via URL

Die zweite Möglichkeit der 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
```



Abbildung 7.3: Projekt Wunderloop im XML-Format

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 7.3).

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.

7.8 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 die Tatsache, dass die traditionelle Verwendung von *link_to* nicht besonders gut mit dem REST-Ansatz korrespondiert, da REST Actionlose 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 der nachfolgenden Tabelle zusammengefasst sind:

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 PUT). 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 7.8.3 und 7.8.2.

Tabelle 7.2: Standard 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

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.

7.8.1 New und Edit

Ein Klick auf den New-Link wird via GET an den 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 7.12.

7.8.2 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 werden Formulare noch einfacher. Egal, ob es sich um ein Neuanlage- oder ein Editier-Formular handelt – der *form_for*-Aufruf ist in beiden Fälle der gleiche:

```
<% form_for(@project) do |f| %>
...
<% end %>
```

Sicher denken Sie jetzt: Moment, werden neue Ressourcen nicht via POST angelegt, während vorhandene Ressourcen via PUT aktualisiert werden? Wenn aber in beiden Fälle der gleiche *form_for*-Aufruf verwendet wird, wie soll der Server da zwischen Neuanlage und Aktualisieren unterscheiden?

Sofern Sie sich für die Antwort auf diese Frage interessieren, sollten Sie diesen Abschnitt zu Ende lesen. Falls nicht, ist es aber auch völlig ausreichend zu wissen, dass Rails genau das erwartete Formular erzeugt, je nachdem, ob Sie der Methode *form_for* ein neues oder ein bereits in der Datenbank vorhandenes Objekt übergeben.

Die Antwort liefert ein Blick in die neue Implementierung von *form_for*. Der Aufruf von *form_for* mit einem neuen Objekt resultiert in folgendem Codeschnipsel:

```
<% form_for :project,
  @project,
  :url => project_path,
  :html => {
    :class => "new_project",
    :id => "new_project"
  } do |f| %>
...
<% end %>
```

Und der Aufruf von *form_for* für ein vorhandenes Objekt entsprechend zu diesem Codeschnipsel:

```
<% form_for :project,
  @project,
  :url => project_path(@project),
```

```

      :html => {
        :method => :put,
        :class => "edit_project",
        :id => "edit_project_1"
      } do |f| %>

      ...
    <% end %>

```

Beide Versionen ähneln der ursprünglichen Aufrufsyntax von *form_for*, unterscheiden sich aber in einem wesentlichen Punkt:

- Für Neuanlage-Formulare wird die Submit-URL vom Helper *project_path* erzeugt.
- Für Editier-Formulare wird die Submit-URL vom Helper *project_path(:id)* erzeugt.

Neuanlage-Formulare

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

```

form_for(:project, @project, :url => projects_path, ... )
=>
<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 unterstützt wird. Die Rails-Lösung ist auch hier die Verwendung des Keys *method* in der *html*-Hash von *form_for*:

```

form_for(:project, @project, :url => projects_path(@project),
  :html => {
    :method => :put,
    ...
  }
=>
<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*. Aber wie gesagt: Eigentlich braucht Sie die Unterscheidung nicht zu interessieren, wenn Sie einfach nur *form_for(@project)* verwenden.

7.8.3 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), :confirm => "Are you sure?",
                                          :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, :confirm => "Are you sure?",
                          :method => :delete

=>
<a href="/projects/1"
  onclick="if (confirm('Are you sure?')) {
    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);
    var s = document.createElement('input');
    s.setAttribute('type', 'hidden');
    s.setAttribute('name', 'authenticity_token');
    s.setAttribute('value', '6dd92f6816f79e5e0167913b6cf1278ec');
    f.appendChild(s); 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.

Seit Rails 2 können Sie *link_to* statt eines Aufrufs von *project_path* auch direkt die entsprechende Modellinstanz übergeben:

```
link_to "Show", project
```

Der URL-Helper *link_to* erkennt die Modellinstanz und erweitert den Aufruf automatisch zu:

```
link_to "Show", project_path(project)
```

Übrigens: Links auf andere Formate können mit Hilfe von Formatted-Path-Methoden erzeugt werden:

```
link_to "XML", formatted_project_path(project, "xml")
```

7.9 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-Anwendung 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)
```

Alternativ können Sie die Modellinstanz auch direkt an den *redirect_to*-Helper übergeben:

```
redirect_to @project
```

Die Methode *redirect_to* erkennt genau wie *link_to* die Modellinstanz und erweitert den Aufruf automatisch zu der zuvor beschriebenen Variante.

Ein Beispiel für den Redirect auf eine Liste von Modellen 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 7.3: ontrack_rest/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
```

7.10 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 stammen. 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 kennengelernt 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)
```

7.10.1 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 *ProjectsControllers* zu routen ist, wenn der zugehörige Request via GET gesendet wurde. Demzufolge muss der Controller über eine Action mit dem Namen *show* verfügen. Dieselbe Konvention gilt für die Actions *index*, *update*, *create*, *destroy*, *new* und *edit*, d.h. REST-Controller müssen diese Methoden implementieren.

7.10.2 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 dasselbe ActiveRecord-Modell *Iteration*, das 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 *OntrackController* geroutet. Ein anderes Beispiel ist die URL

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

die auf die *show*-Action des *OntrackController*s 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
```

7.11 Verschachtelte Ressourcen

Richtig interessant wird REST-basierte Entwicklung bei der Verwendung von verschachtelten Ressourcen⁵. 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.

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

Verschachtelte Ressourcen sind eng gekoppelte Ressourcen im Sinne einer Eltern-Kind-Beziehung. Im Kontext von Rails sind dies Modelle, die z.B. in einer 1:N-Beziehung zueinander stehen, wie beispielsweise Projekte und Iterationen in Ontrack. Controller sind 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 beschreiben. Los geht es mit der Generierung der neuen Ressource *Iteration* sowie der Erzeugung der zugehörigen Datenbanktabelle *iterations*:

```
$ ruby script/generate scaffold 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 7.4: ontrack_rest/app/models/project.rb

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

Listing 7.5: ontrack_rest/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 Helpers 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 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, müssen die generierten *resources*-Einträge für *Project* und *Iteration* in *config/routes.rb* ersetzt werden:

```
map.resources :iterations
map.resources :projects

# wird zu

map.resources :projects, :has_many => [ :iterations ]
```

Die bereits bekannten Routen und Helper für *Project* bleiben bestehen. Darüber hinaus macht der Eintrag *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 *IterationsControllers*, 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 aneinandergehängt werden, wie z.B. der Aufruf der *show*-Action verdeutlicht:

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

7.11.1 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 7.6: `ontrack_rest/app/controllers/iterations.controller.rb`

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

  respond_to do |format|
    format.html # index.html.erb
    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 7.7: ontrack_rest/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. Dies bedeutet aber auch, dass neben *index* auch die Actions *create* (siehe Abschnitt 7.11.3) und *update* angepasst (siehe Abschnitt 7.11.4) werden müssen.

7.11.2 Neue Helper für verschachtelte Ressourcen

Neben den Routen erzeugt der neue resource-Eintrag in *config/routes.rb* eine Reihe neuer Helper für Iterationen:

```
project_iteration_path(:project_id, :iteration_id)
project_iterations_path(:project_id)
new_project_iteration_path(:project_id)
edit_project_iteration_path(:project_id, :iteration_id)
```

Die Namen der neuen Helper bringen die verschachtelte Natur von Iterationen zum Ausdruck, indem der Name der umschließenden Ressource entweder als erstes oder als zweites Wort im Helper-Namen verwendet wird. Beispielsweise wird die Liste aller Iterationen eines Projekts vom Helper *project.iterations_path* erzeugt. Es wird deutlich, dass es sich hier um die Iterationen eines Projekts handelt.

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 *ProjectsControllers*, alle Iterationen des gewählten Projekts anzeigt:

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

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

Listing 7.8: ontrack_rest/app/views/projects/index.rhtml

```
...
<% for project in @projects %>
  <tr>
    <td><%=h project.name %></td>
    <td><%=h project.desc %></td>
    <td>
```

```

    <%= link_to "Iterations", project_iterations_path(project) %>
  </td>
  <td><%= link_to "Show", project_path(project) %></td>
  <td><%= link_to "Edit", project %></td>
  <td><%= link_to "Destroy", project,
    :confirm => "Are you sure?", :method => :delete %></td>
</tr>
<% end %>
...

```

Die Folge der geänderten Namen 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 7.9: ontrack.rest/app/views/iterations/index.html.erb

```

...
<% 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 %></td>
    <td><%= link_to "Edit", edit_iteration_path(iteration) %></td>
    <td><%= link_to "Destroy", iteration,
      :confirm => "Are you sure?", :method => :delete %></td>
  </tr>
<% end %>
...

```

Die Links werden so nicht mehr funktionieren, da im Falle von *Show* und *Destroy* gar kein expliziter Helper und für *Edit* noch der alte Helper verwendet wird. Die notwendige Änderung sieht wie folgt aus:

Listing 7.10: ontrack.rest/app/views/projects/index.html.erb

```

...
<% for iteration in @iterations %>
  <tr>
    <td><%=h iteration.name %></td>
    <td><%=h iteration.start %></td>
    <td><%=h iteration.end %></td>
    <td><%= link_to "Show",
      project_iteration_path(iteration.project,
        iteration) %></td>
    <td><%= link_to "Edit",
      edit_project_iteration_path(iteration.project,
        iteration) %></td>
    <td><%= link_to "Destroy",
      project_iteration_path(iteration.project,
        iteration),

```



```

        :confirm => "Are you sure?",
        :method => :delete %></td>

</tr>
<% end %>
...

```

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

```

project_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.

7.11.3 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 *ProjectsControllers* für jedes angezeigte Projekt um einen *New Iteration*-Link erweitert:

Listing 7.11: `ontrack_rest/app/views/projects/index.html.erb`

```

...
<% for project in @projects %>
  <tr>
    <td><%=h project.name %></td>
    <td><%=h project.desc %></td>
    <td><%= link_to "Iterations",
                  project_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_project_iteration_path(project) %></td>
  </tr>
<% end %>
...

```

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

```

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

```

Der Link wird auf die *new*-Action des *IterationsControllers* 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 *IterationsControllers* zur Verfügung und kann dort im Helper *project_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 7.12: `ontrack_rest/app/views/iterations/new.html.erb`

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

Die Übergabe von `params[:project_id]` in `project_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 => project_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 7.13: `ontrack_rest/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 project_iteration_url(
9                           @iteration.project,
10                          @iteration) }
11       format.xml { render :xml => @iteration,
12                           :status => :created, :location => @iteration }
13     else
14       format.html { render :action => "new" }
15       format.xml { render :xml => @iteration.errors,
16                           :status => :unprocessable_entity }
17     end
18   end
19 end
```

In Zeile 3 wird das Projekt explizit zugewiesen. Außerdem wird in Zeile 8 der Helper *project_iteration_url* unter Angabe der Projekt- und Iterations-Id verwendet.

Damit das Zufügen neuer Iterationen jetzt auch wirklich funktioniert, müssen Sie die Links *Edit* und *Back* im *Show*-View des *IterationsControllers* ändern und anstelle der Standard-Helper für Iteration die Helper für verschachtelte Iteration einsetzen:

Listing 7.14: ontrack_rest/app/views/iterations/show.html.erb

```
...
<%= link_to "Edit", edit_project_iteration_path(@iteration.project,
                                              @iteration) %>

<%= link_to "Back", project_iterations_path(@iteration.project) %>
```

Dieser View wird nämlich nach der Neuanlage gerendert und produziert einen Fehler, wenn die vom Scaffold-Generator erzeugten Helper verwendet werden.

7.11.4 Bearbeiten existierender Iterationen

Zum Bearbeiten von Iterationen sind einige Anpassungen in den generierten Quellen notwendig. Als Erstes wird der *edit*-View des *IterationsControllers* angepasst. Dem *form_for*-Helper wird als *:url*-Parameter die Path-Methode für verschachtelte Iterationen übergeben:

```
form_for(@iteration,
  :url => project_iteration_path(@iteration.project, @iteration),
  :html => { :method => :put }) do |f| %>
```

Außerdem müssen im selben View die Links *Show* und *Back* entsprechend angepasst werden, d.h. aus

```
<%= link_to "Show", @iteration %> |
<%= link_to "Back", iterations_path %>
```

wird

```
<%= link_to "Show", project_iteration_path(@iteration.project,
                                              @iteration) %> |
<%= link_to "Back", project_iterations_path(@iteration.project) %>
```

Ein ähnliche Änderung ist in der vom Formular aufgerufenen *update*-Action durchzuführen. Auch hier wird in Zeile 7 die URL-Methode für verschachtelte Iterationen verwendet:

Listing 7.15: ontrack_rest/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 project_iteration_url(
```

```

8             @iteration.project,
9             @iteration) }
10     format.xml { head :ok }
11   else
12     format.html { render :action => "edit" }
13     format.xml { render :xml => @iteration.errors,
14                     :status => :unprocessable_entity }
15   end
16 end
17 end

```

Die Create- und Update-Views und -Actions sind nach diesen Änderungen so weit wieder lauffähig, dass Iterationen angelegt und bearbeitet werden können. Durchforsten Sie sicherheitshalber noch einmal den *IterationsController* und dessen Views nach Path- und URL-Helfern, die noch nicht über das Präfix *project_* verfügen, und ersetzen Sie diese durch ihr verschachteltes Äquivalent, inklusive der als ersten Parameter benötigten *project.id*.

7.12 Eigene Actions

Der *resources*-Eintrag in der Routing-Datei erzeugt benannte Routen und Helfer für CRUD-Actions. Doch wie sieht es mit Routen und Helfern 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 7.16: *ontrack_rest/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 *ProjectsControllers* benötigt:

Listing 7.17: *ontrack_rest/app/views/projects/index.html.erb*

```

<% for project in @projects %>
  <tr>

```

```

<td><%=h project.name %></td>
<td><%= link_to "Show", 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 PUT 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 PUT aufgerufen werden, so dass der *resources*-Eintrag wie folgt angepasst werden muss:

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

Der neue *resources*-Eintrag sorgt dafür, dass ein Close-Request nur via HTTP-PUT gesendet werden darf. 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:

```
link_to "Close", close_project_path(project), :method => :put
```

Entscheidend ist hier die Übergabe der richtigen HTTP-Methode mit Hilfe des Parameters *:method*. Nur so wird die zugehörige Route gefunden und an die noch zu implementierende *close*-Action weitergeleitet:

Listing 7.18: *ontrack_rest/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."
    end
  end
end

```

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

```

        format.html { redirect_to projects_path }
        format.xml { head 500 }
      end
    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 bleibt 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)

```

7.12.1 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.

7.13 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

```

Sie können eigene Formate definieren, indem Sie diese als neue Formate in der Konfigurationsdatei `config/initializers/mime_types.rb` registrieren:

```
Mime::Type.register_alias "text/html", :iphone
```

Anschliessend kann die `show`-Action des `ProjectsControllers` so erweitert werden, dass Projekte auch in einem abgespeckten HTML-View auf Ihrem iPhone angezeigt werden:

Listing 7.19: `ontrack_rest/app/controllers/projects_controller.rb`

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

  respond_to do |format|
    format.iphone # show.iphone.erb
    ...
  endend
end
```

Zum Testen fehlt jetzt noch ein iPhone-kompatibler HTML-View:

Listing 7.20: `ontrack_rest/app/views/projects/show.iphone.erb`

```
Name: <%=h @project.name %>
```

Und schon können Sie von jetzt an auch unterwegs in Ihrer Projektdatenbank stöbern:

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

7.14 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 `ProjectsControllers` 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 rumprobieren wollen, um herauszukriegen, warum der Link nicht funktioniert. Eine Variante ist ein Eintrag in der Layout-Datei des `ProjectsControllers` `projects.rhtml`:

Listing 7.21: ontrack_rest/app/views/layouts/projects.rhtml

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

Ein Klick auf den Link wird auf die *destroy*-Action des *ProjectsControllers* 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 7.22: 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 namens *destroy.rjs*, das wie folgt anzulegen ist:

Listing 7.23: 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 *ProjectsControllers* funktioniert, müssen die enthaltenen Tabellenzeilen um eine eindeutige Id erweitert werden:

Listing 7.24: ontrack/app/views/projects/index.rhtml

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

Die hier beschriebene Erweiterung des *ProjectsControllers* 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 dieselbe 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.

7.15 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.⁷

```
$ rake
...
Started
EEEEEE.....
```

Die gute Nachricht: Alle Unit-Tests und der funktionale Test *ProjectsControllerTest* laufen noch. Die schlechte: 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 grundsätzlich nicht stimmt. In unserem Fall liegt die Ursache für die Fehler auf der Hand. Der Testcase wurde vom Scaffold-Generator für Iterationen ohne zugehöriges Projekt generiert. Da wir aber manuell eingreifen und Iterationen Projekten zugeordnet sowie 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 7.25: *ontrack_rest/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: Erstens wird die falsche Path-Methode verwendet, und zweitens haben wir 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:

⁷ Achtung: Anlegen der Testdatenbank *ontrack_test* nicht vergessen, sofern nicht bereits geschehen.

```
assert_redirected_to project_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.

7.16 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 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:

```
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 Service spezifiziert. Die Klasse *Project* abstrahiert den clientseitigen Teil des Web Service 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.

7.17 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.

Kapitel 8

Internationalisierung

Wir gehen davon aus, dass viele unserer Leser vor allem deutschsprachige bzw. internationalisierte Anwendungen entwickeln wollen und daher wissen möchten, welche Möglichkeiten Rails diesbezüglich bietet. Deshalb widmen wir diesem Thema ein eigenes Kapitel.

Rails beinhaltet derzeit noch keine eigene Unterstützung für die Internationalisierung von Web-Anwendungen. Dies liegt zum einen daran, dass Ruby selbst von Haus aus keinen Support bietet, und zum anderen an der Tatsache, dass sich bisher keine Lösung als Quasi-Standard durchsetzen konnte. Es ist gut möglich, dass es nie „die“ Standardlösung für Rails geben wird, da viele Lösungen denkbar sind, die nicht für jedes Problem passen.

Das Thema Internationalisierung ist sehr umfangreich, und sämtliche Lösungen und Aspekte zu beschreiben würde ein eigenes Buch erfordern. Es gibt mittlerweile eine Reihe von Bibliotheken für die Internationalisierung von Rails-Anwendungen. Die Seite <http://wiki.rubyonrails.com/rails/pages/InternationalizationComparison/> liefert einen Überblick.

Wir geben Ihnen in diesem Kapitel eine Einführung in die Bibliothek *Ruby-GetText*, die wir auch selbst schon verwendet haben. Als Alternative steht Ihnen u.a. das Plugin *Globalize-rails*¹ zur Verfügung, das recht häufig eingesetzt wird. Zum Zeitpunkt der Niederschrift dieses Buches gab es aber keine stabile Version für Rails 2.0, sodass wir die Einführung weggelassen haben. Vielleicht beschreiben wir das Plugin zu einem späteren Zeitpunkt in Form eines PDFs, das Sie dann auf der Webseite zum Buch finden.

Wir konzentrieren uns vor allem auf die Internationalisierung von Texten sowie Datumsformaten und Zahlen am konkreten Beispiel von *OnTrack*. Die Checkliste in Abschnitt 8.3 führt weitere Punkte auf, die Sie ggf. berücksichtigen müssen.

¹ <http://wiki.globalize-rails.org/wiki>

8.1 Internationalisierung oder Lokalisierung?

Vor dem Einstieg in die Details der Mehrsprachigkeit wollen wir zunächst die Begriffe *Internationalisierung* und *Lokalisierung* genauer bestimmen.

Internationalisierung bedeutet die Auslegung eines Programms auf dessen mehrsprachige Nutzung. Konkret sollte ein internationalisiertes Programm keinerlei hart kodierte Informationen bezüglich seiner Nutzung in einem bestimmten Land enthalten.

Hingegen bedeutet Lokalisierung die Konfiguration eines internationalisierten Programms für die Verwendung einer bestimmten regionalen bzw. kulturellen Umgebung, z.B. Deutschland.

8.2 Lokalisierung und Codes

Um die richtige Lokalisierung der Anwendung für einen bestimmten Benutzer bereitzustellen, müssen Sie wissen, welche Sprache der Benutzer ausgewählt hat.

8.2.1 Language Tag

Die vom Anwender ausgewählte Sprache wird über das so genannte *Language Tag*² nach RFC 3066 festgelegt. Dieser besteht aus dem zweibuchstabigen Sprachcode, z.B. *en* für Englisch, und einem optionalen zweibuchstabigen Ländercode, auch Regionalcode genannt, z.B. *GB* für Großbritannien. Der Sprachcode wird dabei nach ISO-639-1 definiert und der Ländercode nach ISO-3166.

Die Definition eines Ländercodes ermöglicht es, die landesspezifischen Gegebenheiten bei der Lokalisierung zu berücksichtigen. So sprechen Briten wie Amerikaner englisch, es gibt aber Unterschiede in der Sprache, Währung, bei den Datumsformaten usw. Um diese in der Anwendung zu berücksichtigen, werden die Einstellungen über den Language Tag *en-GB* für Großbritannien und *en-US* für die Vereinigten Staaten festgelegt.

8.2.2 Locale

Alle Einstellungen zur Sprache, zum Zeichensatz, zu den Zahlen-, Währungs-, Datums- und Zeitformaten usw. werden in der sogenannten *Locale* definiert. Viele Bibliotheken definieren daher eine Klasse *Locale* und legen die aktuell zu verwendende Locale über den Language Tag fest, z.B. per *Locale.set(„en-GB“)*.

Ihre Anwendung kann auf verschiedene Arten an den Language Tag gelangen. Er kann als Request-Parameter oder als Cookie an die Anwendung übergeben werden oder in der URL codiert sein. Ebenso ist die Speicherung in der Session denkbar.

Wichtig ist, dass sich Ihre Anwendung nicht auf die Angabe eines Language Tags verlässt. Sie muss immer eine Locale für einen Language Tag definieren, der keinen

² Vgl. <http://www.w3.org/International/articles/language-tags/>

Ländercode definiert, und eine Default-Locale anbieten, falls gar kein Language Tag übergeben wird.

Am Beispiel von Englisch und Großbritannien wäre der Ablauf wie folgt: Der Anwendung wird der Language Tag *en-GB* übergeben, und existiert eine Locale hierfür, so wird diese verwendet. Gibt es nur eine Locale für die Sprache *en*, dann wird diese genutzt. Welche landesspezifischen Einstellungen (z.B. *GB* oder *US*) Sie in diesem Fall definieren, bleibt Ihnen überlassen. Gibt es gar keine passende Locale, so ist die Default-Locale zu verwenden.

Erhält die Anwendung nur den Sprachcode *en*, so nehmen Sie die entsprechende Locale. Ist keine passende vorhanden oder wurde kein Language Tag übergeben, so ist die Default-Locale zu verwenden (meist für Englisch).

Viele Bibliotheken stellen diese Funktionalität bereit, so auch die beiden in diesem Kapitel vorgestellten.

Weitere Informationen zum Thema Locale und den Language Tag finden Sie auch unter http://de.wikipedia.org/wiki/ISO_639, <http://de.wikipedia.org/wiki/ISO-3166-1-Kodierliste> und <http://de.wikipedia.org/wiki/Locale>.

8.3 Checkliste

Um eine Anwendung zu internationalisieren, bedarf es mehr als nur der Übersetzung von Texten. Im Folgenden haben wir eine Checkliste zusammengestellt, die die wesentlichen Punkte bei der Internationalisierung einer Anwendung aufführt. Prüfen Sie Ihre Anwendung anhand der Liste, und erweitern Sie diese bei Bedarf:

- Meldungen und Hilfetexte
- Zahlen (z.B. 12,345.67 oder 12.345,67)
- Datums- und Zeitangaben
- Währungen und Maßeinheiten (z.B. cm, inch)
- Sortierung und Wortvergleiche
- Töne und Tonstücke bzw. Videos mit Sprache
- Farben, Grafiken und Icons
- Telefonnummern und Anschriften
- Anrede und Titel
- Seitenlayout (z.B. Lesen von rechts nach links)

8.4 Internationalisierung vorbereiten

Für die Internationalisierung einer Rails-Anwendung ist diese zunächst auf den Zeichensatz Unicode³ umzustellen. Dieser definiert alle aktuell bekannten Zeichen und stellt daher den Standard für die Internationalisierung einer Anwendung auch außerhalb von Rails und Ruby dar. Die Zeichensatzcodierung von Unicode erfolgt mit UTF-8⁴, UTF-16 usw., wobei UTF-8 am weitesten verbreitet ist.

Für unsere Rails-Anwendung speichern wir alle Dateien (Modelle, Controller, Views, Konfigurationsdateien etc.) in UTF-8. Besteht die Notwendigkeit, die Dateien außerhalb des Editors in UTF-8 zu konvertieren, so kann Ihnen das Programm *iconv*⁵ bzw. dessen Ruby-Implementierung helfen:

```
# iconv_test.rb
require 'iconv'
# => R<FC>by => R<C3><BC>by
result = Iconv.conv("ISO-8859-1", "UTF-8", "Rüby")
puts result
```

Wir ändern die Defaultcodierung der Datenbanktabellen in UTF-8. In den Migrationsskripten geben wir dazu beim Anlegen einer Tabelle über den Parameter *options* Optionen an die Datenbank. Im Falle von MySQL in etwa wie folgt:

```
class Initial < ActiveRecord::Migration
  OPTIONS = "DEFAULT CHARSET=utf8"

  def self.up
    create_table "projects", :options => OPTIONS do |t|
      ...
    create_table "iterations", :options => OPTIONS do |t|
      ...
    end
  end
end
```

Zusätzlich muss die Übertragung der Daten aus der Datenbank per UTF-8 erfolgen. Hierzu fügen wir für MySQL den Eintrag *encoding: utf8* zur *config/database.yml* hinzu.⁶

```
# config/database.yml
development:
  adapter: mysql
  database: ontrack_development
  host: localhost
  username: rails
  password: ...
  encoding: utf8
```

³ <http://de.wikipedia.org/wiki/Unicode>

⁴ <http://de.wikipedia.org/wiki/UTF8>

⁵ <http://www.gnu.org/software/libiconv/>

⁶ Im Falle anderer Datenbanken lautet der Eintrag ggf. anders. Siehe auch <http://wiki.rubyonrails.org/rails/pages/HowToUseUnicodeStrings>

Des Weiteren müssen wir auch Ruby selbst die Umstellung auf UTF-8 mitteilen. Ruby und seine Methoden arbeiten nämlich von Haus aus Byte-orientiert. Dadurch beträgt z.B. die Länge des Wortes *Rüby* 5 Zeichen und nicht 4, weil das *ü* intern durch zwei Bytes repräsentiert wird:

```
puts "Rüby".length # => 5
```

Ruby unterstützt zwar die Verwendung anderer Zeichensätze, doch gilt dies leider nicht für die meisten Methoden aus der Core- bzw. Standard-Bibliothek. Das ändert sich hoffentlich mit künftigen Ruby-Versionen. Bis dahin können wir nur teilweise Abhilfe schaffen und fügen folgende Zeilen in die Datei *config/environment.rb* ein:

```
$KCODE = 'u'
require 'jcode'
```

Über die Variable *\$KCODE* wird der Zeichensatz für Ruby festgelegt. In unserem Falle setzen wir die Variable auf *u* für UTF-8. Außerdem laden wir die Standardbibliothek *jcode*, die eine modifizierte Version der Ruby *String*-Klasse enthält. Diese sorgt dafür, dass bestimmte Methoden wie *delete* oder *chop* mit dem per *\$KCODE* definierten Zeichensatz umgehen können. Darüber hinaus stehen Methoden wie *jlength* oder *jcount* bereit:

```
$KCODE = 'u'
require 'jcode'
puts "Rüby".length # => 5
puts "Rüby".jlength # => 4
```

Damit Browser die in UTF-8 ausgelieferten Views richtig interpretieren, muss in jedem View der Zeichensatz angegeben werden. Wir erreichen dies durch folgende Definition im Layout *app/views/layouts/projects.html.erb*:

```
# app/views/layouts/projects.html.erb
<head>
  <meta http-equiv="Content-type" value="text/html; charset=utf-8">
  ...
</head>
```

Damit sind die allgemeinen Vorbereitungen abgeschlossen, und wir können eine der beiden im Folgenden beschriebenen Lösungen einsetzen. Weitere interessante Informationen zu Unicode und Rails finden Sie auch im Rails-Wiki unter <http://wiki.rubyonrails.org/rails/pages/HowToUseUnicodeStrings>. Es empfiehlt sich, den Unicode-Zeichensatz von vornherein für jede Anwendung zu nutzen. Damit ersparen Sie sich später viel Ärger.

8.5 Ruby Gettext-Package

Das *Ruby Gettext-Package* von Masao Mutoh implementiert eine Ruby-Schnittstelle zum weit verbreiteten Gettext-Paket von GNU⁷. Im Prinzip werden bei Gettext alle zu übersetzenden Texte in die Methoden *gettext()* oder *_()* eingeschlossen, z.B.:

⁷ <http://www.gnu.org/software/gettext/>


```
_('I love Rails')
```

Zur Laufzeit wird die entsprechende Übersetzung in Abhängigkeit von der aktuell gesetzten Sprache aus einer Textdatei ermittelt. Im Falle von Deutsch findet Gettext z.B. die Übersetzung *Ich liebe Rails*. Das Paket ist auch unabhängig von Rails für reine Ruby-Anwendungen einsetzbar.

8.5.1 Installation

Die Installation des Ruby Gettext-Packages erfolgt über *gem*:

```
$ sudo gem install gettext
Attempting local installation of 'gettext'
...
Select which gem to install for your platform (i686-darwin8.6.1)
...
Building native extensions. This could take a while...
...
Successfully installed gettext-1.90.0
```

Im Falle von Linux oder Mac wählen Sie die letzte *ruby*- und im Falle von Windows die letzte *mswin32*-Version. Für Linux und Mac benötigen Sie einen C-Compiler, da native Bindings übersetzt werden. Alternativ können Sie alles auch ohne *gem* installieren. Schauen Sie dazu unter <http://www.yotabanana.com/hiki/ruby-gettext.html> nach.

Außerdem benötigen Sie eine plattformspezifische Version von Gettext selbst. Entsprechende Ressourcen finden Sie unter <http://www.gnu.org/software/gettext/>. Unter Mac empfehlen wir die Installation über Fink⁸ per *apt-get install gettext*.

Für Windows gibt es unter <http://gnuwin32.sourceforge.net/packages/gettext.htm> ebenfalls einige Portierungen. Nach der Installation sollten u.a. die Programme *msgfmt* und *msgmerge* existieren und auf der Kommandozeile ausführbar sein.

8.5.2 Texte übersetzen

Die Übersetzungen werden bei Gettext in Dateien gehalten. Anfangs wird eine *pot*-Datei mit Hilfe eines Programms erstellt, das alle Texte der Anwendung aufammelt und in die Datei einfügt. Aus der *pot*-Datei wird pro Sprache eine *po*-Datei erstellt und mit den Übersetzungen gefüllt.

Die Abkürzung *po* steht für Portable Object⁹ und repräsentiert eine Textdatei, die von Menschen gelesen und verändert werden kann. Aus einer *po*-Datei wird mit Hilfe eines weiteren Programms eine *mo*-Datei erzeugt. Die Abkürzung *mo* steht für Machine Object und repräsentiert eine Binärdatei mit allen Texten. Gettext arbeitet zur Laufzeit der Anwendung auf Basis der *mo*-Dateien, da dies performanter ist. Schauen wir uns die Schritte näher an.

⁸ <http://fink.sourceforge.net>

⁹ Und *pot* für Portable Object Template

pot-Datei erstellen

Für die Übersetzung Ihrer Anwendung müssen die Texte aus den Views, Controllern und Modellen in die pot-Datei gelangen. Dazu sind zwei Schritte notwendig. Im ersten sind alle zu übersetzenden Texte in die Methoden *gettext()* oder *_()* einzuschließen. Im zweiten Schritt werden diese Texte mit Hilfe eines Programms aus der Anwendung extrahiert und in die pot-Datei eingefügt.

Für ein erstes Beispiel nehmen wir uns den *index*-View für Projekte vor, der wie folgt geändert wird. Beachten Sie, dass Sie alle Texte in die Tags *<%=* und *%>* einschließen:

Listing 8.1: app/views/projects/index.html.erb

```
<h1><%= _('Listing projects') %></h1>
...
<%= for project in @projects %>
  <td><%= link_to _('Show'), ...
  <td><%= link_to _('Edit'), ...
  <td><%= link_to _('Destroy'), ...
      :confirm => _('Are you sure?') %>
  <td><%= link_to _('Add Iteration'), ...
...

```

Mit Hilfe des Ruby-Programms *rgettext* werden alle in *gettext* und *_* eingeschlossenen Texte extrahiert. Rufen Sie testweise einmal folgenden Befehl auf:

```
$ rgettext app/views/projects/index.html.erb
```

Sie erhalten auf der Konsole die folgende Ausgabe (durch Angabe der Option *-o Datei* können Sie diese auch in eine Datei schreiben):

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
...
#: app/views/projects/index.html.erb:1
msgid "Listing projects"
msgstr ""
...
#: app/views/projects/index.html.erb:19
msgid "Destroy"
msgstr ""
...

```

Für jeden Text wurde jeweils ein Eintrag bestehend aus *msgid* und *msgstr* angelegt. Findet Gettext später z.B. den Eintrag *_('Destroy')* im View, so sucht er diesen in der Textdatei und liefert die Übersetzung aus *msgstr* zurück. Gibt es keinen solchen Eintrag, wird der Schlüssel selbst als Text zurückgegeben.

Das Programm *rgettext* kann aber immer nur eine Datei verarbeiten, und die Angabe der Zieldatei muss explizit erfolgen. Um in einem Schritt alle Dateien der Anwendung zu verarbeiten, automatisieren wir den gesamten Prozess durch folgenden Rake-Task, den wir unter *lib/tasks/gettext.rake* erstellen:

Listing 8.2: lib/tasks/gettext.rake

```
require 'gettext/utils'
require 'gettext/rails'

namespace :gettext do
  desc "Update po-files."
  task :updatepo do
    GetText.update_pofiles("ontrack",
      Dir.glob("app/**/*.{rb,html.erb}"), "ontrack 1.0")
  end
end
```

Hierdurch werden alle Dateien mit der Endung *rb* und *html.erb* im Verzeichnis *app/* verarbeitet. Gegebenenfalls müssen Sie weitere Dateiendungen hinzufügen. Der Aufruf erfolgt über:

```
$ rake gettext:updatepo
(in .../ontrack)
po/ontrack.pot
```

Als Ergebnis entsteht im Verzeichnis *po/* die Datei *ontrack.pot* mit allen Texten der Anwendung. Diese Datei enthält bereits ein Reihe von Einträgen, obwohl wir zunächst nur den View *app/views/projects/index.html.erb* vorbereitet haben. Das liegt daran, dass Ruby Gettext im Kontext von Rails auch die Tabellen- und Spaltennamen der Datenbank aufammelt (siehe Abschnitt 8.5.7). Für den Anfang konzentrieren wir uns auf den View und betrachten die anderen Einträge nicht. Ebenso wird hier exemplarisch nur der *index*-View für Projekte betrachtet. Die Übersetzung der anderen Dateien überlassen wir Ihnen.

Übersetzung in po-Dateien

Für jede Sprache ist nun aus der pot-Datei eine po-Datei zu erstellen, in die die Übersetzungen eingefügt werden. Dazu kopieren wir die Datei in ein entsprechendes Verzeichnis pro Sprache. Der Verzeichnisname ergibt sich aus dem Sprach- und Regionalcode (z.B. *en* und *GB*). Benötigen Sie den Regionalcode nicht, dann können Sie diesen auch weglassen und das Verzeichnis nur nach dem Sprachcode benennen. Achten Sie auf die neue Endung *.po*:

```
$ mkdir po/de_DE po/en_GB
$ cp po/ontrack.pot po/de_DE/ontrack.po
$ cp po/ontrack.pot po/en_GB/ontrack.po
```

Hinweis: Sie können an dieser Stelle auch das Gettext-Programm *msginit* verwenden, wenn dieses in Ihrer Gettext-Distribution enthalten ist. Es nimmt Ihnen die im Folgenden aufgeführten Schritte ab.

Die Bearbeitung der po-Dateien kann mit einem gewöhnlichen Texteditor erfolgen, solange dieser UTF-8 unterstützt. Die von Ruby-Gettext erzeugten pot- bzw. po-Dateien sind zu GNU Gettext kompatibel. Daher können Sie alle Werkzeuge und Programme aus GNU Gettext auch für Ruby-Gettext nutzen. Für die Übersetzung

steht Ihnen mit *poEdit*¹⁰ ein Programm mit graphischer Oberfläche zur Verfügung. Vielleicht liegt Ihnen oder Ihrem Übersetzer das näher.

Wir nutzen den Editor und löschen in den Dateien *po/de_DE/ontrack.po* und *po/en_GB/ontrack.po* die Zeile *#fuzzy*. Zeilen mit einem solchen Kommentar kennzeichnen Einträge mit möglicherweise inkorrekt übersetzter Übersetzung. Diese können z.B. entstehen, wenn es beim Zusammenführen zweier po-Dateien mit dem Programm *msgmerge*¹¹ zu unterschiedlichen Einträgen bei gleicher *msgid* kommt.

Außerdem prüfen wir die Werte für Content-*, die den Zeichensatz UTF-8 und eine 8-Bit-Codierung definieren müssen:

```
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

Ebenso ist die Definition für die Plural-Formen einer Sprache notwendig. Gültige Werte finden Sie unter http://www.gnu.org/software/gettext/manual/html_mono/gettext.html#DOCF5. Im Falle von Deutsch und Englisch lautet der Eintrag:

```
"Plural-Forms: nplurals=2; plural=(n != 1);\n"
```

Anschließend können wir die Übersetzungen einfügen, z.B. wie folgt:

```
# po/de_DE/ontrack.po
...
msgid "Listing projects"
msgstr "Projektliste"
...
```

Sind alle Texte in der Anwendung in Englisch, so müssen Sie die englische po-Datei *po/en_GB/ontrack.po* nicht unbedingt füllen bzw. überhaupt erstellen. (Das gilt natürlich auch für Deutsch oder jede andere Sprache, in der die Originaltexte geschrieben sind.) Wie bereits erwähnt, liefert Ruby-Gettext immer den Ursprungstext selbst zurück, falls es keinen Eintrag dazu findet.

Aktualisierung der po-Dateien

Werden der Anwendung nachträglich weitere Texte hinzugefügt, erfolgt die Aktualisierung der pot- und po-Dateien durch einen erneuten Aufruf von *updatepo*:

```
$ rake gettext:updatepo
(in .../ontrack)
po/ontrack.pot
..... done.
po/de_DE/ontrack.po
..... done.
po/en_GB/ontrack.po
..... done.
```

¹⁰ <http://www.poedit.net>

¹¹ Siehe hierzu unter <http://www.gnu.org/software/gettext/manual>.

Dabei werden jetzt auch die po-Dateien aktualisiert, sodass wir uns diesen Schritt sparen können. Nach dem Einfügen der Übersetzung müssen die mo-Dateien neu erzeugt werden.

mo-Dateien erstellen

Pro po-Datei müssen wir eine mo-Datei erstellen, die von Gettext zur Laufzeit der Anwendung genutzt wird. Dies kann auch mit Hilfe von poEdit geschehen. Im Beispiel nutzen wir aber einen weiteren Rake-Task, den wir in die Datei *lib/tasks/gettext.rake* einfügen:

```
# lib/tasks/gettext.rake
require 'gettext/utils'
...
namespace :gettext do
  ...
  desc "Create mo-files for L10n"
  task :makemo do
    GetText.create_mofiles(true, "po", "locale", "%s/LC_MESSAGES")
  end
end
```

Die Übersetzung aller po-Dateien erfolgt dann über den folgenden Aufruf:

```
$ rake gettext:makemo
(in .../ontrack)
po/de_DE/ontrack.po -> locale/de_DE/LC_MESSAGES/ontrack.mo
po/en_GB/ontrack.po -> locale/en_GB/LC_MESSAGES/ontrack.mo
```

Die mo-Dateien landen im Verzeichnis *locale/* und hier wiederum aufgeteilt nach Sprache. Nach der Aktualisierung der mo-Dateien ist kein Neustart des Servers notwendig, da die Dateien neu eingelesen werden.

Ruby-Gettext und Suchpfade

Ruby-Gettext sucht die mo-Dateien in dem im rake-Task angegebenen Ordner und in allen Verzeichnissen des Rails-Suchpfads (Globale Variable *\$:*). Vorsicht also, falls Sie in einem der Verzeichnisse eine mo-Datei abgelegt haben, die nicht zur Anwendung gehört!

Weitere Verzeichnisse können über die Methode *add_default_locale_path* hinzugefügt werden, z.B.

```
class Date
  add_default_locale_path(
    ".../locale/{%locale}/LC_MESSAGES/{%name}.mo")
  ...
end
```

Die Platzhalter *locale* und *name* werden dabei durch die Sprache (z.B. *de_DE*) und die Textdomäne (den Namen der mo-Datei, z.B. *ontrack*) ersetzt (siehe auch Listing 8.5).

8.5.3 Die Schritte im Überblick

Im Folgenden haben wir die Schritte als Überblick und Gedankenstütze aufgeführt:

1. Alle Texte in den Views, Controllern, Modellen internationalisieren, d.h. in die Methode *gettext* oder *_* einschließen (z.B. `<%= _('Speichern') %>`).
2. Alle Texte aus Punkt 1 in eine pot-Datei extrahieren, indem Sie den Rake-Task *gettext:updatepo* ausführen.
3. Aus der pot-Datei für jede Sprache eine po-Datei erstellen.
4. Die po-Datei mit den Übersetzungen füllen.
5. Die von Gettext zur Laufzeit der Anwendung genutzten mo-Dateien erstellen, d.h. den Rake-Task *gettext:makemo* ausführen.
6. Bei neuen Texten die Schritte wiederholen, wobei unter 2. auch die po-Dateien automatisch aktualisiert werden und somit 3. entfällt.

8.5.4 Anwendung und Gettext zusammenbringen

Nachdem wir die Übersetzungen erstellt haben, soll unsere Anwendung sie auch nutzen. Dazu definieren wir die so genannte Textdomäne zentral im *Application-Controller*, wodurch sie für alle Controller und damit die gesamte Anwendung gilt:

Listing 8.3: Gettext im Controller initialisieren

```
# app/controllers/application.rb
require 'gettext/rails'

class ApplicationController < ActionController::Base
  ...
  init_gettext "ontrack"
  ...
end
```

Die Methode *init_gettext* sorgt für die Verbindung zwischen den mo-Dateien (der Textdomäne *ontrack*) und allen Controllern, Views und Modellen der Anwendung. Neben der Textdomäne können der Zeichensatz (Default UTF-8) und der Content-Type (Default text/html) definiert werden:

```
class ApplicationController < ActionController::Base
  ...
  init_gettext "ontrack", {:charset => "UTF-8",
                          :content_type => "text/html"}
  ...
end
```

Sie können auch pro Controller eine eigene Textdomäne (und damit pot-Datei) definieren, z.B. wenn Sie den Admin-Bereich nicht oder anders übersetzen wollen.

Bleibt nur noch zu klären, wie die Anwendung die aktuell zu verwendende Sprache erkennt. Dazu sucht Ruby-Gettext der Reihe nach im Request-Parameter *pa-*

`rams[:lang]`, im Eintrag `lang` im `QUERY.STRING`, im Cookie `lang`, im Request-Header (`request.env["HTTP_ACCEPT_LANGUAGE"]`), oder es wird die Default-Sprache verwendet (`Locale.set.default`).

Da Sie ein deutsches Buch lesen, gehen wir davon aus, dass Ihr Browser über `HTTP_ACCEPT_LANGUAGE` Deutsch als Sprache überträgt. Der Aufruf der URL `http://localhost:3000/projects` sollte daher die in Abbildung 8.1 aufgeführte Seite der Projektliste in deutscher Sprache darstellen.

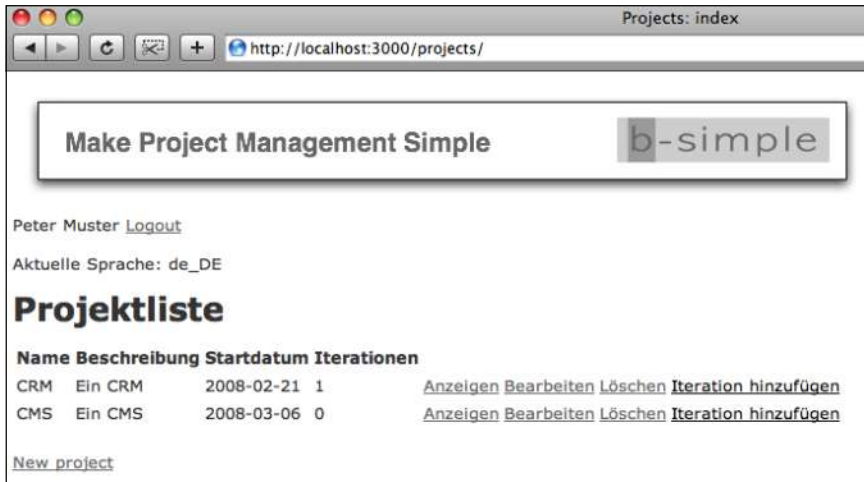


Abbildung 8.1: Deutsches OnTrack mit Ruby-Gettext

Der Aufruf der URL `http://localhost:3000/projects?lang=de-de` führt zum selben Ergebnis. Hingegen stellt der Aufruf `http://localhost:3000/projects?lang=de` die Seite in Englisch dar. Das liegt daran, dass Gettext nun die Übersetzung im Verzeichnis `locale/de` und nicht `locale/de.DE/` sucht. Gibt es nur eine `mo`-Datei unter `locale/de`, so findet Gettext diese auch dann, wenn die Sprache mit Ländercode (`de.DE`) übergeben wurde.

8.5.5 Lokalisierte Templates

Innerhalb von Rails unterstützt Ruby-Gettext auch lokalisierte Templates, indem es zuerst nach Views mit dem Namen `index_de.DE.html.erb` und `index.de.html.erb` sucht, bevor es den View `index.html.erb` nutzt. Dies ist von Vorteil, wenn die Seite viel statischen Text enthält. Hier kann es sinnvoll sein, eine neue Seite (View) pro Sprache zur Verfügung zu stellen, statt die Texte in die `po`-Datei auszulagern. Dieses Feature funktioniert auch für Action Mailer Templates (vgl. Kapitel 9).

8.5.6 Controller

Die Übersetzung von Texten in einem Controller erfolgt analog zu den Views. Wir internationalisieren exemplarisch die Texte der `create`-Action des `ProjectController`:

```
# app/controllers/projects_controller.rb
def create
  @project = Project.new(params[:project])
  ...
  if @project.save
    flash[:notice] = _('Project was successfully created.')
    ...
  end
end
```

Dieser Text landet beim Ausführen unserer rake-Tasks ebenfalls in den po-Dateien und kann entsprechend übersetzt werden.

8.5.7 Modelle

Diese Modelle enthalten zu übersetzende Texte, z.B. zur Validierung. Diese schließen wir ebenfalls in die Methode `_()` ein:

Listing 8.4: app/models/iteration.rb

```
class Iteration < ActiveRecord::Base
  ...
  validates_presence_of :name
  validates_length_of :name, :minimum => 5
  ...
  protected

  def validate
    if end_date <= start_date
      errors.add(:end_date,
        _("End date must be after the start date"))
    end
  end
  ...
end
```

Fehlermeldungen zur Validierung

Für Rails stellt Ruby-Gettext bereits eine Übersetzung der Fehlermeldungen für die Validierungsmethoden (vgl. Abschnitt 4.14.1) zur Verfügung (siehe Verzeichnis *po/* der Ruby-Gettext-Installation). Geben Sie testweise einmal keinen und einen zu kurzen Iterationsnamen ein. Die Meldungen erscheinen bereits auf Deutsch. Auch der Attributname und die Mindestanzahl der Zeichen werden angegeben.

Wenn Sie für die Validierungen eigene Texte nutzen wollen, so müssen Sie die Methode `N_` verwenden:

```
class Iteration < ActiveRecord::Base
  ...
  validates_presence_of :name,
    :message => N_("The value of %{fn} can't be empty!")
  validates_length_of :name, :minimum => 5,
```



```

      :message => N_("The value of %{fn} is too short (min %d)")
    ...
  end

```

Ruby-Gettext bietet, wenn nötig, den Platzhalter `%{fn}` für den *field name* und `%d` für die Anzahl. Nach dem obligatorischen `rake gettext:updatepo` fügen wir die Übersetzungen in `po/de_DE/ontrack.po` ein:

```

# po/de_DE/ontrack.po
...
msgid "The value of %{fn} can't be empty!"
msgstr "Der Wert für %{fn} darf nicht leer sein!"
...
msgid "The value of %{fn} is too short (min %d)"
msgstr "Der Wert für %{fn} ist zu kurz (min %d)"

```

Anschließend rufen wir wieder `rake gettext:makemo` auf, und die darauf folgende Testeingabe zeigt die eigenen Fehlermeldungen mit Feldnamen und Mindestlänge aus Abbildung 8.2.

Aktuelle Sprache: de_DE

Editing Iteration

3 Fehler haben verhindert, dass dieser Iteration gespeichert werden konnte

Folgende Felder haben Probleme verursacht:

- Der Wert von Name darf nicht leer sein!
- Der Wert von Name ist zu kurz! (min 5)
- End date Das Enddatum muss größer als das Startdatum sein

Name:

Start Date: 2008 March 11

End Date:

2008 March 9

Abbildung 8.2: Fehleranzeige in Deutsch

Tabellen- und Spaltennamen

Per Default ermittelt Ruby-Gettext über die Modelle auch alle Tabellen- und Spaltennamen aus der Datenbank und fügt diese in die pot- bzw. po-Dateien ein. Die Einträge haben dabei das Format *Klassenname|Feldname*, also zum Beispiel:

```
# po/de_DE/ontrack.po
...
# app/models/iteration.rb:-
msgid "Iteration|Description"
msgstr ""
```

Eine Übersetzung der Texte ist in diesem Fall nur für den letzten der durch den Balken (—) getrennten Teile notwendig. Wir schreiben also:

```
msgid "Iteration|Description"
msgstr "Beschreibung"
```

Des Weiteren unterstützt Ruby-Gettext die Übersetzung von Feldnamen, die nicht im Datenbankschema enthalten sind. Dies kommt bei Rails z.B. bei der Validierung der Passwortwiederholung vor. Hierbei handelt es sich um ein Pseudoattribut, das für die Eingabe und Validierung existiert, aber nicht in der Datenbanktabelle vorhanden ist. Hier nutzen Sie die Methode *N_()* wie folgt:

```
class Person < ActiveRecord::Base
  ...
  N_("Person|Password")
  N_("Person|Password confirmation")
  ...
end
```

Um die Verarbeitung von Feldern zu unterbinden, nutzen Sie die Methode *untranslate*:

```
class Iteration < ActiveRecord::Base
  untranslate :description
  ...
end
```

Soll keiner der Spaltennamen einer Tabelle übersetzt werden, verwenden Sie im Modell die Methode *untranslate_all*:

```
class Person < ActiveRecord::Base
  untranslate_all
  ...
end
```

8.5.8 Dynamische Texte und Pluralisierung

Ein nicht unerheblicher Teil der zu übersetzenden Texte hat dynamischen Inhalt. Die Übersetzung geschieht in diesem Fall mit Hilfe der Ruby String-Methode *%*. Als ein-

faches Beispiel nutzen wir den View *app/views/iterations/edit.html.erb* und fügen exemplarisch den folgenden Text unterhalb der Überschrift ein¹²:

```
<h1>Editing Iteration</h1>
<p>
  <%= _('Add iteration no %d to project: %s') %
    [@iteration.project.iterations.length+1,
     @iteration.project.name] %>
</p>
<%= error_messages_for 'iteration' %>
...
```

Wir aktualisieren die po-Dateien per *rake gettext:updatepo* und fügen die Übersetzung in *po/de_DE/ontrack.po* ein:

```
# po/de_DE/ontrack.po
...
msgid "Add iteration no %d to project: %s"
msgstr "Iteration Nr. %d zum Projekt '%s' hinzufügen"
...
```

Der String selbst enthält an den entsprechenden Stellen Platzhalter, wie zum Beispiel *%s* für Werte des Typs String. Die möglichen Platzhalter entnehmen Sie bitte der API-Dokumentation zur Ruby-Methode *%* bzw. *Kernel.sprintf*. Nach dem Aufruf *rake gettext:makemo* wird der Text in der jeweiligen Sprache mit dynamischen Anteilen angezeigt.

Reihenfolge dynamischer Werte

Die Reihenfolge der dynamischen Werte ist im Programmcode fest kodiert und für jede Sprache gleich. Die Sprache kann jedoch eine Satzstellung erfordern, bei der die dynamischen Werte vertauscht werden müssen. Angenommen, die Übersetzung von „Add iteration no %d to project: %s“ lautet „Zum Projekt %s die Iteration Nr. %d hinzufügen.“. Da die dynamischen Werte immer in der Reihenfolge *Iterationsnummer* und *Projektname* kommen, haben wir hier ein Problem. Vielleicht wirkt das Beispiel etwas konstruiert, aber die Konstellation ist durchaus möglich.

Als Lösung erlaubt Gettext die Angabe von Positionen. Die Übersetzung des Textes enthält dabei die Positionen in der Form *Position\$* und lautet „Zum Projekt %2\$s die Iteration Nr. %1\$d hinzufügen“. Dadurch wird der erste dynamische Parameter (die Iterationsnummer) an die Stelle des zweiten Platzhalters eingefügt und der Projektname an die des ersten.

Pluralisierung

Ein anderes Problem bei der Internationalisierung dynamischer Texte sind die unterschiedlichen Singular- und Pluralformen eines Satzes. Die Ausgabe der Iterationen zu einem Projekt lautet je nach Anzahl z.B.:

```
Das Projekt hat keine Iteration
```

¹²Die Anpassung der anderen Texte lassen wir der Einfachheit halber weg.

```
Das Projekt hat eine Iteration
Das Projekt hat %d Iterationen
```

In diesem Fall verwenden wir die Methode `n_`, der wir den Text für die Ein- und Mehrzahl und die aktuelle Anzahl übergeben. Wir ersetzen den ursprünglichen Eintrag im View `app/views/iterations/edit.html.erb` wie folgt:

```
<% count = @iteration.project.iterations.length %>
<%= n_('Project %s has one iteration',
      'Project %s has %d iterations', count+1) %
    [@iteration.project.name, count] %>
```

Die Aktualisierung der po-Dateien per `rake gettext:updatepo` fügt folgenden Eintrag in `po/de.DE/ontrack.po` ein:

```
msgid "Project %s has one iteration"
msgid_plural "Project %s has %d iterations"
msgstr[0] ""
msgstr[1] ""
```

Hier kommt der Eintrag im Kopf der Datei ins Spiel:

```
"Plural-Forms: nplurals=2; plural=(n != 1);\n"
```

Unsere Übersetzung lautet:

```
msgid "Project %s has one iteration"
msgid_plural "Project %s has %d iterations"
msgstr[0] "Projekt %s hat eine Iteration."
msgstr[1] "Projekt %s hat %d Iterationen."
```

Nach dem erneuten Aufruf von `rake gettext:makemo` zeigt OnTrack den Text in Abhängigkeit der Anzahl der Iterationen an.

Um auch den Text „Das Projekt hat keine Iteration“ anzuzeigen, müssen Sie den Programmcode wie folgt anpassen, da dies Ruby-Gettext nicht automatisch erledigen kann:

```
<% count = @iteration.project.iterations.length %>
<% if count == 0 %>
  <%= _('Project has no iteration') %>
<% else %>
  <%= n_('Project %s has one iteration',
        'Project %s has %d iterations', count+1) %
    [@iteration.project.name, count] %>
<% end %>
```

8.5.9 Datum, Zahlen und Währungen

Gettext bietet keine Unterstützung für die Formatierung von Daten und Zahlen oder Währungen. Wir haben uns daher einige Hilfsmethoden geschrieben, von denen wir Ihnen die zur Formatierung des Datums kurz präsentieren.

Ruby erlaubt die Erweiterung aller Klassen. Dies machen wir uns für die Klasse *Date* zu Nutze und erweitern diese um die Methode *f*:

```
# lib/extend/i18n/date.rb
require 'date'
require File.dirname(__FILE__) + "/ilocale"

class Date
  FORMAT = { 'de_DE' => "%d.%m.%Y", 'de' => "%d.%m.%Y" }

  def f(format = nil)
    format = format || FORMAT[I18n.locale.code]
    format ? self.strftime(format) : self
  end
end
```

Die Methode liefert das Datum in Abhängigkeit des übergebenen Formats zurück. Ist keines angegeben, ermitteln wir das Format aus der Hash *FORMAT* in Abhängigkeit der aktuellen Sprache (z.B. *de_DE* oder *de*), die wir überall über die Hilfsmethode *I18n.locale.code* erhalten. Im Falle von Ruby-Gettext ist diese wie folgt implementiert:

```
# lib/extend/i18n/ilocale.rb
module I18n
  def self.code
    I18n.current.to_posix
  end
end
```

Im Falle anderer Bibliotheken ändern wir hier die Implementierung und können alle anderen Methoden ohne Änderung weiterverwenden.

Die Dateien *date.rb* und *ilocale.rb* werden im Verzeichnis *lib/extend/i18n* gespeichert und am Ende der Datei *environment.rb* eingebunden:

```
...
Dir["#{RAILS_ROOT}/lib/extend/i18n/*.rb"].each {|file| require file}
```

Nun können wir die Methode wie folgt nutzen, z.B. im View *app/views/projects/show.html.erb*. Die Ausgabe erfolgt dann im Format *TT.MM.YYYY*:

```
<p>
  <b><%= _('Start date') %></b>
  <%=h @project.start_date.f %>
</p>
```

Im Falle der Ruby-Klasse *Time* gehen wir analog vor, nehmen aber die Uhrzeit in das Format mit auf:

```
require 'time'
require File.dirname(__FILE__) + "/ilocale"

class Time
```

```

silence_warnings do
  FORMAT = {
    'de_DE' => "%d.%m.%Y %H:%M:%S",
    'de'     => "%d.%m.%Y %H:%M:%S"
  }
end

def f(format = nil)
  format = format || FORMAT[I18n.locale]
  format ? self.strftime(format) : self
end
end

```

8.5.10 Wochen- und Monatsnamen

Die Helper-Methode `date_select` (vgl. Abschnitt 6.2) verwendet u.a. die Wochen- und Monatsnamen aus der Ruby-Klasse `Date`. Die Namen sind in der Klasse über Konstanten und in englischer Sprache definiert und müssen von uns lokalisiert werden. Ruby erlaubt die Redefinition von Konstanten, sodass wir unsere `Date`-Klasse aus `lib/extend/i18n/date.rb` wie folgt anpassen:

Listing 8.5: Erweiterung der `Date`-Klasse

```

# lib/extend/i18n/date.rb
require 'date'
require 'gettext/rails'
require File.dirname(__FILE__) + "/i18n/locale"

class Date
  include GetText # für _()

  # Pfad auf Übersetzungsdatei.
  add_default_locale_path(
    "#{RAILS_ROOT}/locale/%{locale}/LC_MESSAGES/%{name}.mo" )

  # Verbinde die Klasse mit Textdatei
  bindtextdomain "ontrack", { :locale => I18n.locale }

  silence_warnings do
    MONTHNAMES = [nil, _('January'), ..., _('December')]
    DAYNAMES    = [_('Sunday'), ..., _('Saturday')]
    ABBR_MONTHNAMES = [nil, _('Jan'), ..., _('Dec')]
    ABBR_DAYNAMES  = [_('Sun'), ..., _('Sat')]
  end

  ...
end

```

Über *add_default_locale_path* landen wir für die Klasse die Textdatei, die wir über *bindtextdomain* definieren. Der Parameter *:locale* definiert die zu verwendende Sprache.

Wir erweitern den rake-Tasks *:updatepo* unter *lib/tasks/gettext.rake*, damit er die Dateien im Verzeichnis *lib/extend/i18n* berücksichtigt:

```
...
namespace:gettext do
  desc "Update po-files."
  task :updatepo do
    GetText.update_pofiles("ontrack",
      Dir.glob("app/**/*.{rb,html.erb}") +
      Dir.glob("lib/extend/i18n/*.rb"),
      "ontrack 1.0")
  end
end
...
end
```

Durch einen erneuten Aufruf von *rake gettext:updatepo* stehen nun auch die Monatsnamen usw. in unserer Textdatei *po/de_DE/ontrack.po*:

```
...
msgid "January"
msgstr "Januar"

msgid "February"
msgstr "Februar"

msgid "March"
msgstr "März"
...
```

Hier fügen wir die deutschen Namen ein und erzeugen die mo-Datei per *rake gettext:makemo*. Nach einem Neustart des Webserver stehen uns die Monatsnamen in deutscher Sprache zur Verfügung. Testen Sie einmal das Hinzufügen einer Iteration, und achten Sie auf den Monatsnamen in der Auswahlbox, wenn Sie die URL um den Parameter *...?lang=de_DE* erweitern.

In Produktion wird die Datei *date.rb* aber nur einmal geladen, und den Konstanten werden die übersetzten Namen nur einmal zugewiesen, und zwar beim Start der Anwendung. Ein Wechsel der Sprache zur Laufzeit der Anwendung würde die Übersetzung der Namen nicht neu einlesen.

Wir helfen uns hier pragmatisch und laden die Klasse bei jedem Request. Dazu definieren wir den Filter *prepare_i18n* im *ApplicationController*. Besser wäre es, die Datei nur beim Wechsel der Sprache neu einzulesen. Diese Verbesserung überlassen wir Ihnen als Aufgabe.

```
class ApplicationController < ActionController::Base
  ...
  before_filter :prepare_i18n
```

```

protected
  def prepare_i18n
    load File.dirname(__FILE__) + "/../lib/extend/i18n/date.rb"
  end
end

```

Reihenfolge von Tag, Monat und Jahr

In Auswahlboxen für das Datum kann die Reihenfolge von Tag, Monat und Jahr je nach Sprache unterschiedlich sein. Die Helper-Methode *date_select* bietet über die Option *:order* die Möglichkeit, die Reihenfolge zu definieren. Wir nutzen dies, indem wir die Methode *date_select* wie folgt neu definieren:¹³

```

# lib/extend/i18n/date_helper.rb
module ActionView::Helpers::DateHelper
  ORDER = {
    'de_DE' => [:day, :month, :year],
    'de' => [:day, :month, :year]
  }
  alias_method :orig_date_select, :date_select

  def date_select(object, method, options = {})
    options[:order] = ORDER[I18n.locale.code]
    orig_date_select(object, method, options)
  end
end

```

Wir überschreiben die Methode und ermitteln aus *ORDER* die Reihenfolge in Abhängigkeit von der aktuellen Sprache. Gibt es keinen Eintrag, wird die Standardreihenfolge von *date_select* verwendet. Aktuell haben wir nur zwei Einträge für Deutsch definiert. Wenn nötig, fügen Sie weitere hinzu. Auch diese Datei legen wir unter dem Namen *date_helper.rb* im Verzeichnis *lib/extend/i18n* ab.

8.5.11 Sortierung

Die Sortierung eines Arrays von Namen muss an die lokalen Gegebenheiten angepasst werden. So sortiert die Ruby-Methode *sort* per Default ein Wort wie „Ägypten“ ans Ende einer Liste von Ländern. Bei einer deutschsprachigen Anwendung würde man den Namen jedoch am Beginn der Liste erwarten.

Gettext bietet keine Unterstützung für die korrekte Sortierung von Wörtern. Hier kann Ihnen vielleicht die Verwendung der *ICU4R*-Bibliothek¹⁴ helfen. Sie definiert Klassen wie z.B. *UCollator*, die auf Basis von Unicode arbeiten und eine korrekte Sortierung nach Sprache und Land ermöglichen.

¹³Diese Erweiterung für *select_month* u.a. lassen wir der Einfachheit halber weg.

¹⁴<http://icu4r.rubyforge.org/>

8.5.12 Zusammenfassung

Der Abschnitt 8.5 konnte nur eine kurze Einführung in die Verwendung von Ruby-Gettext und Gettext darstellen. Sollten Sie sich für Gettext entscheiden, empfehlen wir Ihnen einen Blick in die Dokumentation <http://www.gnu.org/software/gettext/>, insbesondere in das Handbuch. Hier erhalten Sie viele Hinweise und Tipps zur richtigen Nutzung, vor allem bezüglich dynamischer Texte und Pluralisierung. Gerade wenn Sie nicht selbst der Übersetzer der Texte sind, ist das Wissen über die richtige Nutzung der Methoden `_` und `n_` notwendig.

Weitere Informationen zur Nutzung von Ruby-Gettext-Bibliothek finden Sie unter der Seite <http://www.yotabanana.com/hiki/ruby-gettext.html> oder auf der Rails-Homepage unter <http://manuals.rubyonrails.com/read/chapter/105>. Ebenso ist die API-Dokumentation im installierten gem-Paket hilfreich.

Denken Sie auch daran, die Datei `lib/tasks/gettext.rake` und die neuen pot- und po-Dateien in Ihre Versionsverwaltung aufzunehmen, sonst war die Mühe umsonst.

Kapitel 9

Action Mailer

Das *Action Mailer* Framework ermöglicht den Versand von E-Mails für Auftragsbestätigungen, Registrierungen, vergessene Passwörter oder Ähnliches. Ebenso können Sie E-Mails empfangen und somit Tickets, Anfragen oder Aufträge verarbeiten. Rails stellt Ihnen auch hier wieder entsprechende Testmöglichkeiten bereit. Wir werden im Folgenden den Versand und Empfang von E-Mails am Beispiel erklären.

9.1 E-Mail-Versand

Für den Versand einer Mail benötigen wir den Text, die üblichen Parameter, wie z.B. den Betreff, den Empfänger und den Absender, sowie eine Möglichkeit, die E-Mail zu versenden. Das Erstellen der E-Mail ist Ihre Aufgabe, den Versand übernimmt das Action Mailer Framework.

9.1.1 E-Mail erstellen

Als erster Schritt ist eine Mailer-Klasse anzulegen, die von der Framework-Basis-Klasse *ActionMailer::Base* erbt. In dieser definieren Sie die Angaben zum Empfänger, Absender usw. sowie, falls vorhanden, die dynamischen Anteile Ihres E-Mail-Textes.

Als Beispiel erstellen wir im Folgenden die Klasse *ProjectStatusMailer*, die den aktuellen Projektstatus an den Projektleiter versendet. Die Klasse erhält eine Methode *status*, in der die Attribute der Mail festgelegt werden. Rails stellt für das initiale Gerüst einen Generator bereit, den wir wie folgt aufrufen:

```
$ ruby script/generate mailer ProjectStatusMailer status
  create  app/views/project_status_mailer
  exists  test/unit/
  create  test/fixtures/project_status_mailer
  create  app/models/project_status_mailer.rb
  create  test/unit/project_status_mailer_test.rb
  create  app/views/project_status_mailer/status.erb
  create  test/fixtures/project_status_mailer/status
```

Eine Mailer-Klasse wird in Rails als Modell verstanden und daher unter *app/models/* abgelegt. Das passt, denn schließlich ermittelt man die dynamischen Daten der Mail (z.B. die Auftragsdaten) aus den Modellen. Neben der Klasse selbst werden ein zugehöriges Template *status.erb* für den E-Mail-Text und entsprechende Unit Test-Komponenten erzeugt. Wie im Folgenden zu sehen ist, erbt die Klasse *ProjectStatusMailer* von *ActionMailer::Base* und definiert die gewünschte Methode *status*. Diese enthält bereits die wesentlichen Attribute einer E-Mail. Eine komplette Übersicht finden Sie im Anhang unter 14.11.

Listing 9.1: *app/models/project_status_mailer.rb*

```
class ProjectStatusMailer < ActionMailer::Base
  def status(sent_at = Time.now)
    @subject      = "ProjectStatusMailer#status"
    @body         = {}
    @recipients   = ""
    @from         = ""
    @sent_on      = sent_at
    @headers      = {}
  end
end
```

Für unser Beispiel erweitern wir die Methode *status*, indem wir ihr das Projekt als Objekt übergeben und damit die E-Mail-Attribute definieren:

Listing 9.2: *app/models/project_status_mailer.rb*

```
class ProjectStatusMailer < ActionMailer::Base
  def status(project, sent_at = Time.now)
    @subject      = "Project Status: #{project.name}"
    @body         = { :project => project }
    @recipients   = project.lead.mail
    @from         = "status@b-simple.de"
    @sent_on      = sent_at
    @headers      = {}
  end
end
```

Statt der Attribute können auch die korrespondierenden Setter-Methoden verwendet werden. Die Methode würde dann wie folgt aussehen:

Listing 9.3: *app/models/project_status_mailer.rb*

```
class ProjectStatusMailer < ActionMailer::Base
  def status(project, sent_at = Time.now)
    subject       "Project Status: #{project.name}"
    body          ({ :project => project })
    recipients    project.lead.mail
    from          "status@b-simple.de"
    sent_on       sent_at
    headers       {}
  end
end
```

Alle in der Hash zum Attribut *body* definierten Werte stehen im Template zur Methode bereit. Action Mailer erzeugt für jeden Eintrag eine Instanzvariable mit dem Namen des Schlüssels. Im Template haben Sie Zugriff auf diese Instanzvariablen und definieren damit die dynamischen Anteile des E-Mail-Textes. Im Beispiel wird die Projektinstanz unter dem Schlüssel *:project* gespeichert und steht uns über die Instanzvariable *@project* zur Verfügung.

Das Template *status.erb* zur Methode hat der Generator unter *app/views/project.status.mailer/* erzeugt. Die Datei hat noch keinen Inhalt und wird für das Beispiel mit folgendem Text gefüllt:

Listing 9.4: *app/views/project.status.mailer/status.erb*

```
Projekt Statusbericht!

Projekt: <%= @project.name %>!
Startdatum: <%= @project.start_date %>
Fertige Tasks: <%= @project.closed_tasks %>
Offene Tasks: <%= @project.open_tasks %>
...
--
OnTrack - Project Management
```

Der Text kann HTML-Code enthalten. In diesem Fall ist der Content-Type¹ der E-Mail in der Methode *status* über das Attribut *content_type* entsprechend auf „*text/html*“ zu setzen:

Listing 9.5: *app/models/project.status.mailer.rb*

```
class ProjectStatusMailer < ActionMailer::Base
  def status(project, sent_at = Time.now)
    ...
    @content_type = "text/html"
  end
end
```

Helper verwenden

Innerhalb des Templates können Sie alle Helper aus Action Pack verwenden, z.B.:

Listing 9.6: *app/views/project.status.mailer/status.erb*

```
...
Projektname: <%= truncate(@project.name, 20) %>
...
```

Ausnahme bilden Helper wie *link_to*, die eine URL aus Controller, Action usw. erzeugen. Dazu ist der aktuelle Request notwendig, den Action Mailer jedoch nicht kennt. Wollen Sie Links in Ihrem Template verwenden, so müssen Sie diese komplett selbst definieren:

¹ Siehe <http://de.wikipedia.org/wiki/Content-Type>

Listing 9.7: `app/models/project_status_mailer.rb`

```
class ProjectStatusMailer < ActionMailer::Base
  def status(project, sent_at = Time.now)
    body({ :project => project,
          :ontrack1 => "http://b-simple.de/ontrack",
          :ontrack2 => ontrack_url(:host => "b-simple.de"),
          :ontrack3 => url_for(:host => "b-simple.de",
                              :controller => "welcome",
                              :action => "index") })

    ...
  end
end
```

Listing 9.8: `app/views/project_status_mailer/status.erb`

```
...
<%= @ontrack1 %>
<%= @ontrack2 %>
<%= @ontrack3 %>
...
```

Beachten Sie, dass die aktuelle Domain (z.B. *b-simple.de*) immer explizit anzugeben ist, da eine relative URL wie */project/index* im E-Mail-Client keinen Effekt hat.

Existiert ein entsprechendes Helper-Modul (im Beispiel *app/helpers/project_status_mailer_helper.rb*), so stehen die dort definierten Methoden im Template ebenfalls zur Verfügung:

Listing 9.9: `app/helpers/project_status_mailer_helper.rb`

```
module ProjectStatusMailerHelper
  def format_date(date)
    ...
  end
end
```

Listing 9.10: `app/views/project_status_mailer/status.erb`

```
...
Startdatum: <%= format_date(@project.start_date) %>
...
```

Wollen Sie Methoden aus anderen Helper-Modulen (z.B. *ApplicationHelper*) verwenden, müssen Sie das jeweilige Modul in *ProjectStatusMailerHelper* einbinden:

Listing 9.11: `app/helpers/project_status_mailer_helper.rb`

```
module ProjectStatusMailerHelper
  include ApplicationHelper
  ...
end
```

Alternativ definieren Sie das Helper-Modul (hier *ApplicationHelper*) direkt im Mailer-Modell:

Listing 9.12: `app/models/project_status_mailer.rb`

```
class ProjectStatusMailer < ActionMailer::Base
  helper :application
  ...
end
```

Partial Views verwenden

Sie können innerhalb der Templates auch Partial Views (vgl. Abschnitt 6.5) verwenden. Im folgenden Beispiel binden wir den Partial View `app/views/projects/_project.erb` ein:

Listing 9.13: `app/views/project_status_mailer/status.erb`

```
...
<%= render(:partial => "projects/project", :object => @project) %>
...
```

Mit der Definition der Attribute und dem Erstellen des Textes sind die Vorbereitungen abgeschlossen.

9.1.2 E-Mail-Objekt erzeugen

Um die erstellte Mail über Action Mailer zu versenden, ist zweierlei notwendig: Zum einen muss die Mail dem Standard Message-Format RFC2822² für Mails entsprechen, und zum anderen muss die Übertragung über einen SMTP³-Server oder ein Sendmail⁴-Programm konfiguriert werden.

Keine Sorge, die Mail wird von Action Mailer automatisch in das Standard Message-Format konvertiert. Dies geschieht mit Hilfe der E-Mail-Bibliothek *TMail*⁵, die mit Rails ausgeliefert wird.

Action Mailer stellt Ihnen hierzu für jede Instanzmethode in Ihrer Mailer-Klasse automatisch die Methoden `create_XXX` und `deliver_XXX` zur Verfügung. Erstere liefert ein E-Mail-Objekt der Klasse *TMail::Mail*, Letztere verschickt ein solches. Für das Beispiel wird die Status-Mail somit über die Methode `create_status` erzeugt.

Um sich das praktisch klarzumachen und die über *TMail* erzeugte Mail anzuschauen, kann Ihnen folgendes Ruby-Programm helfen, das Sie ggf. Ihren Bedürfnissen anpassen oder allgemein gültiger schreiben können:

Listing 9.14: `test/tmail.rb`

```
require File.dirname(__FILE__) + "../config/environment"
project = Project.find(:first)
mail = ProjectStatusMailer.create_status(project)
puts "\n#{mail.encoded}"
```

² Siehe <http://www.faqs.org/rfcs/rfc2822.html>

³ Siehe <http://de.wikipedia.org/wiki/SMTP>

⁴ Siehe <http://de.wikipedia.org/wiki/Sendmail> oder <http://www.sendmail.org>

⁵ Siehe <http://i.loveruby.net/en/prog/tmail.html>

Es wird ein Projekt aus der Datenbank gelesen und der Methode *create_status* übergeben. Diese ruft intern unsere Methode *ProjectStatusMailer#status* auf und erhält darüber die Attribute der Mail. Aus diesen und dem Text im Template erzeugt *create_status* ein Objekt von *TMail::Mail* und liefert es zurück.

Die Ausgabe sieht dann in etwa wie folgt aus:

```
$ ruby test/tmail.rb

Date: Sun, 2 Oct 2005 13:17:27 +0200
From: status@b-simple.de
To: chief@b-simple.de
Subject: Project Status: Erstes Rails Projekt
Content-Type: text/plain; charset=utf-8

Projekt Statusbericht!

Projekt: Erstes Rails Projekt!
Startdatum: 2005-09-18
Fertige Tasks: 42
Offene Tasks: 17

--
OnTrack - Project Management
```

In der Anwendung wird es typischerweise ein Modell oder einen Controller geben, aus dem die E-Mail in Abhängigkeit der Geschäftslogik versendet wird, z.B. die Klasse *Project*:

Listing 9.15: app/models/project.rb

```
class Project < ActiveRecord::Base
  ...
  private
    def send_status_mail
      ProjectStatusMailer.deliver_status self
    end
end
```

9.1.3 E-Mail versenden

Um die Mail (als *TMail::Mail*-Objekt) zu versenden, benötigen wir eine Übertragungsmöglichkeit. Wie bereits erwähnt, unterstützt Action Mailer den Versand von Mails über einen SMTP-Server oder über Sendmail.

Per Default verwendet Action Mailer den Versand über SMTP. Entscheiden Sie sich für Sendmail, so müssen Sie Action Mailer über die folgende Konfigurationseinstellung davon in Kenntnis setzen. Erstellen Sie am besten eine eigene Datei *config/initializers/initializer_mailer.rb* für die Mail-Konfiguration:

Listing 9.16: config/initializers/initializer_mailer.rb

```
ActionMailer::Base.delivery_method = :sendmail
```

Der Versand über Sendmail funktioniert zur Zeit nur bei einer vorhandenen Sendmail-Installation unter `/usr/sbin/sendmail` auf Ihrem Rechner. Hinweise zur Installation finden Sie auf der Sendmail-Homepage <http://www.sendmail.org>.

Der Versand über SMTP ist die einfachere Variante, da heute fast jeder E-Mail-Provider einen solchen Zugang anbietet. Action Mailer benötigt hierzu die Verbindungsdaten zum SMTP-Server, die über die Variable `ActionMailer::Base.smtp_settings` definiert und z.B. in der Datei `config/initializers/mailer.rb` abgelegt werden:

Listing 9.17: `config/initializers/mailer.rb`

```
ActionMailer::Base.smtp_settings = {  
  :address => "smtp.web.de",  
  :port => 25,  
  :domain => "benutzername@web.de",  
  :authentication => :login,  
  :user_name => "benutzername",  
  :password => "passwort"  
}
```

Über `:address` und `:port` sprechen Sie den Server an. Dieser erwartet ggf. eine Identifikation des Client-Rechners, häufig über den Domainnamen (Eintrag `domain`). Erfordert die Kommunikation mit dem Server keine Authentifizierung, ist für `:authentication` der Wert `:plain` anzugeben. Andernfalls erfolgt die Anmeldung über `:login` mit Benutzername und Passwort. Zusätzlich steht Ihnen die Option `:cram_md5`⁶ zur Verfügung. Beachten Sie, dass die Übertragung der Daten unverschlüsselt geschieht, da es zurzeit keine Unterstützung für die gesicherte Übertragung seitens Action Mailer gibt.

Stellen Sie doch einmal die Daten des SMTP-Servers Ihres Providers ein, und verschicken Sie wie beschrieben eine Mail. Schließlich freut man sich ja über eine nette Mail im eigenen Postfach. Sie müssen dazu die Methode `create_status` durch `deliver_status` im Beispielprogramm `test/tmail.rb` ersetzen.

Listing 9.18: `test/tmail.rb`

```
...  
# mail = ProjectStatusMailer.create_status project  
mail = ProjectStatusMailer.deliver_status project  
...
```

Tritt beim Versand ein Fehler auf, wird die E-Mail trotzdem angezeigt, da Ausnahmen per Default unterdrückt werden. Sie können dieses Verhalten ändern, indem Sie das Werfen einer Ausnahme über `ActionMailer::Base.raise_delivery_errors` aktivieren. In diesem Fall wird statt der E-Mail die Ausnahme angezeigt.

Listing 9.19: `config/initializers/mailer.rb`

```
ActionMailer::Base.raise_delivery_errors = true
```

⁶ Siehe <http://de.wikipedia.org/wiki/Cram-md5>

Ist z.B. der SMTP Server nicht korrekt in den *ActionMailer::Base.smtp_settings* definiert, erscheint die folgende Fehlermeldung (sie ist nicht besonders aussagekräftig, aber immerhin eine Meldung):

```
$ ruby test/tmail.rb
/usr/lib/ruby/1.8/net/protocol.rb:83:in initialize:
  getaddrinfo: No address associated with nodename
  (SocketError)
```

Wenn alles richtig konfiguriert wurde, steht dem Versand der ersten Mail über Action Mailer nichts mehr im Wege. Wenn's geklappt hat, dann herzlichen Glückwunsch!

9.1.4 Testen

Beim Unit Test zu Ihrem Mailer haben Sie die Möglichkeit, den von der Anwendung erzeugten Inhalt der Mail mit dem erwarteten abzugleichen. Hierzu verwenden Sie die vom Generator erzeugte Testklasse *ProjectStatusMailerTest* in *test/unit/project_status_mailer_test.rb*. Der bereits enthaltene Code lädt das Fixture *test/fixture/project_status_mailer/status* mit dem erwarteten Mail-Inhalt und vergleicht diesen mit dem von der Methode *create_status* erzeugten. Sie müssen daher im Fixture den exakten Text der Mail einfügen und bei jeder Änderung des Templates entsprechend korrigieren. Diese Art Test ist insgesamt mühevoll und fehleranfällig.

Wenn Sie auf eine exakte Überprüfung verzichten können, so testen Sie nur die korrekte Definition der Attribute und der dynamischen Anteile im Mail-Text. Ein einfacher Test in *ProjectStatusMailerTest* sieht wie folgt aus:

Listing 9.20: test/unit/project_status_mailer_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class ProjectStatusMailerTest < ActionMailer::TestCase
  tests ProjectStatusMailer
  def test_status
    mail = ProjectStatusMailer.create_status projects(:rails)
    assert_equal "Project Status: Erstes Rails Projekt",
      mail.subject
    assert_equal "chief@b-simple.de", mail.to[0]
    assert_equal "status@b-simple.de", mail.from[0]
    assert_match "Projekt: #{projects(:rails).name}", mail.body
    ...
  end
end
```

Den Versand der E-Mail aus einem Modell oder Controller testen Sie über den zugehörigen Unit bzw. Functional Test. Im Listing 9.21 verschickt die Controller-Action z.B. die Mail und führt eine Weiterleitung auf die Bestätigungssseite durch:

Listing 9.21: `app/controllers/project_status_mailer_controller.rb`

```
class ProjectStatusMailerController < ApplicationController
  def send_status_mail
    project = Project.find params[:id]
    ProjectStatusMailer.deliver_status project
    redirect_to :action => "show_send_confirmation"
  end
end
```

Der entsprechende Test prüft, ob die Action erfolgreich durchlaufen und tatsächlich eine Mail verschickt wurde:

Listing 9.22: `test/functional/project_status_mailer_controller_test.rb`

```
class ProjectStatusMailerControllerTest < ActionController::TestCase
  def test_send_status_mail
    sent_mails = ActionMailer::Base.deliveries
    sent_mails.clear

    get :send_status_mail, :id => projects(:rails)
    assert_redirected_to :action => "show_send_confirmation"
    assert_emails 1
    assert_equal "Project Status: Erstes Rails Projekt",
                 sent_mails[0].subject
    assert_equal "chief@b-simple.de",
                 sent_mails[0].to[0]
    assert_equal "status@b-simple.de",
                 sent_mails[0].from[0]
  end
end
```

Wenn Sie in diesen Unit und Functional Tests auch gleich die Attribute und die dynamischen Werte im Mail-Text prüfen, können Sie sich den Mailer-Test aus 9.20 ggf. sparen.

Damit nicht bei jedem Testlauf E-Mails verschickt werden, simuliert Action Mailer den Versand in der Testumgebung. Hierzu ist die Versandart in `config/environment/test.rb` auf `:test` gesetzt:

Listing 9.23: `config/environments/test.rb`

```
config.action_mailer.delivery_method = :test
```

Die E-Mails werden dann in dem Array `ActionMailer::Base.deliveries` gespeichert, auf das Sie in Ihrer Test-Klasse Zugriff haben.

9.1.5 Tipp zum zuverlässigen E-Mail-Versand

Das Versenden von E-Mails kann neben Programmierfehlern auch durch technische Probleme verhindert werden. So ist möglicherweise der SMTP-Server kurzzeitig nicht erreichbar, oder es liegt ein Netzwerkproblem vor.

Schicken Sie z.B. in Ihrem Online-Shop nach der Bestellung eine Bestätigungs-E-Mail und kommt es zu einem Fehler, erhält der Kunde statt der erwarteten Bestätigungsseite möglicherweise die Fehlerseite (500.html). Das schafft nicht unbedingt Vertrauen. Selbst wenn Sie die Ausnahme abfangen und dem Anwender eine benutzerfreundliche Fehlermeldung anzeigen, die Bestätigungs-E-Mail wurde nicht versendet.

Sie können Action Mailer veranlassen, in solch einem Fall keine Ausnahme zu werfen (siehe `ActionMailer::Base.raise_delivery_errors` oben). Dann erfahren Sie aber auch nie, dass es zu Fehlern gekommen ist.

Um den Versand zuverlässiger zu gestalten, können Sie die E-Mail, statt sie direkt aus dem Modell oder Controller zu versenden, in der Datenbank speichern. Der Versand erfolgt über einen externen Prozess (z.B. als Cron-Job oder per Background-DRb⁷), der periodisch auf auszuliefernde Mails prüft. Das Abschicken wird so lange wiederholt, bis es funktioniert oder ein Grenzwert an fehlerhaften Versuchen überschritten wird. Klappt der Versand, wird die Mail aus der Datenbank gelöscht.

9.1.6 Multipart E-Mails

Action Mailer unterstützt den Versand von Mails mit mehreren Teilen von unterschiedlichem Content-Type (Text, Bild, PDF usw.), so genannte Multipart Mails. Hierbei kann die Angabe der Teile explizit oder implizit erfolgen.

Explizit angegebene Teile

Im Falle der expliziten Angabe sind die einzelnen Teile der Mail in der Mailer-Methode anzugeben. Soll im Beispiel neben dem Status auch ein Report in Form eines PDFs versendet werden, so sieht der Aufruf wie folgt aus:

Listing 9.24: `app/models/project_status_mailer.rb`

```
# app/models/project_status_mailer.rb
class ProjectStatusMailer < ActionMailer::Base
  def status_with_report(project, sent_at = Time.now)
    subject      "Project Status: #{project.name}"
    recipients   project.lead.mail
    from         "status@b-simple.de"
    sent_on      sent_at

    part "text/plain" do |p|
      p.body = render_message("status", :project => project)
      p.transfer_encoding = "base64"
    end
    part "application/pdf" do |p|
      p.body = File.read("report.pdf")
      p.content_disposition = "attachment"
      p.transfer_encoding = "base64"
      p.filename = "report.pdf"
    end
  end
end
```

⁷ <http://backgroundrb.rubyforge.org/>

```

    end
  end
end

```

Der Eintrag für *body* entfällt und wird durch den ersten mit *part* eingeleiteten Block ersetzt. Über die Methode *render_message* wird der E-Mail-Text aus dem Template *status.erb* erzeugt und anschließend per Base64⁸ kodiert.

Als zweiter Teil der Mail folgt das PDF, welches aus der Datei *report.pdf* relativ zum Projektverzeichnis gelesen wird. Die Anordnung (Disposition) des Textes innerhalb der Mail wird über die Methode *content_disposition* als Anhang („*attachment*“) definiert. Wird keine Disposition angegeben, wird per Default *inline* verwendet. Der Teil wird dann, wie im Beispiel der Statustext, direkt in der Mail angezeigt. Der beim Speichern des Anhangs vorgeschlagene Dateiname wird über *filename* definiert. Weitere Attribute (z.B. der Content-Type) können durch entsprechend gleichlautende Methoden definiert werden (siehe dazu Abschnitt 14.11 im Anhang).

Die Block-Variante ist flexibel, weil Sie in den Block noch weitere Funktionalität einsetzen können. Alternativ kann der Methode *part* auch eine Hash mit Werten übergeben werden:

```

part({ :content_type => "text/plain",
      :body => render_message("status", :project => project ),
      :transfer_encoding => "base64" })

```

Wenn Sie im Testprogramm *tmail.rb* den Aufruf von *create_status* in *create_status_with_report* ändern, liefert der erneute Aufruf folgende Ausgabe:

```

Date: Thu, 13 Oct 2005 13:41:03 +0200
From: status@b-simple.de
To: chief@b-simple.de
Subject: Project Status: Rails Projekt
Content-Type: multipart/mixed; boundary=mimepart_434...

--mimepart_434e638c4dde0_3ed723c814b
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: Base64
Content-Disposition: inline

UHJvamVrdCBTdGF0dXNiZXJpY2h0IQoKUHJvamVrdDogUmFpbHMg...

--mimepart_434e638c4dde0_3ed723c814b
Content-Type: application/pdf; name=report.pdf
Content-Transfer-Encoding: Base64
Content-Disposition: attachment; filename=report.pdf

JVBERi0xLjMKJcTl8uXrp/Og0MTGCjIgMCEvYmoKPDwgL0xlbmd0...

```

⁸ Siehe <http://de.wikipedia.org/wiki/Base64>

Implizit angegebene Teile

Statt die Teile der E-Mail explizit in der Methode zu definieren, können diese auch implizit durch die Templates bestimmt werden. Der Content-Type des jeweiligen Teils wird dabei über den Dateinamen definiert.

Soll im Beispiel die Mail aus einem reinen Text und einem HTML-Teil bestehen, so sind die entsprechenden Templates *status.text.plain.erb* und *status.text.html.erb* im Verzeichnis *app/views/project.status.mailer/* anzulegen. Die Mail besteht dann aus zwei Teilen vom Content-Type *"text/plain"* und *"text/html"*.

Per Default werden die Teile in der Reihenfolge *"text/plain"*, *"text/enriched"* und *"text/html"* eingefügt. Die Reihenfolge der Anzeige im E-Mail Client ist genau anders herum, sodass die Anzeige von HTML vor der von Text erfolgt.

Die Grundeinstellung kann durch die Variable *implicit_parts_order* explizit in der Mailer-Methode gesetzt werden, z.B.:

Listing 9.25: *app/models/project.status.mailer.rb*

```
def status(project, sent_at = Time.now)
  @subject = "Project Status: #{project.name}"
  ...
  @implicit_parts_order = [ "text/html" , "text/plain" ]
  ...
end
```

Angabe des Anhangs vereinfachen

Um den Versand einer Mail mit Anhang etwas zu vereinfachen, bietet Action Mailer statt der Methode *part* die Methode *attachment*. Der Aufruf und die Parameter sind identisch, doch sind bei *attachment* die Attribute für die Disposition mit *"attachment"* und *transfer.encoding* mit *"base64"* bereits vorbelegt. Dadurch wird dieser Teil der Mail als Anhang festgelegt und per Base64 kodiert.

Der Aufruf über *attachment* sieht für das Beispiel dann wie folgt aus:

```
attachment "application/pdf" do |p|
  fn = "report.pdf"
  p.body = File.read(fn)
  p.filename = fn
end
```

9.2 E-Mail-Empfang

Action Mailer ermöglicht auch die Verarbeitung eingehender E-Mails. Wie das prinzipiell funktioniert, wird in diesem Abschnitt beschrieben.

9.2.1 E-Mail empfangen

Um eine eingehende Mail zu verarbeiten, muss die Anwendung zunächst einmal an die Mail gelangen. Von wo aus und wie die Mail an die Anwendung weitergeleitet werden kann, ist je nach Betriebssystem, eingesetzten Programmen und Konfiguration verschieden. Alle Kombinationen zu beschreiben, würde zu weit führen. Wir zeigen Ihnen daher, wie eine Mail prinzipiell an die Anwendung übergeben wird, und überlassen es Ihnen, die hierfür notwendigen Umgebungen in Ihrem konkreten Fall herzustellen.

Im Beispiel rufen wir eine Mail von einem POP3⁹-Postfach ab und übergeben diese an die OnTrack-Anwendung. Das Beispiel simuliert eine an OnTrack geschickte Mail mit einem neuen Task. Aus den Daten der Mail wird der Task ermittelt und als neuer Eintrag in der Datenbank gesichert.

Aus Sicht von Action Mailer wird eine Mailer-Klasse benötigt, die die Methode *receive* definiert. Diese erhält als Parameter immer ein Objekt der Klasse *TMail::Mail*, aus der die entsprechenden Informationen ermittelt werden. Ihre Aufgabe ist es, die Mail an die Methode *receive* zu übergeben und entsprechend zu verarbeiten.

Es wird zunächst die Mailer-Klasse *TaskMailReceiver* generiert:

```
$ ruby script/generate mailer TaskMailReceiver
...
```

Die Methode *receive* wird im Beispiel wie folgt implementiert:

Listing 9.26: app/model/task_mail_receiver.rb

```
class TaskMailReceiver < ActionMailer::Base
  def receive(mail)
    task = Task.new
    task.name = mail.subject
    ...
    task.save
  end
end
```

Sie nimmt die Mail entgegen, ermittelt daraus die Informationen und speichert den neuen Task in der Datenbank. Der typische Weg von Daten über den View und Controller ins Modell ist für eine Mail wenig sinnvoll, weshalb E-Mails direkt an die Methode *receive* im Modell übergeben werden. Zu Demonstrationszwecken nutzen wir folgendes kleines Hilfsprogramm:

Listing 9.27: test/getpop3.rb

```
require "net/pop"
require File.dirname(__FILE__) + "../config/environment"

POP3_SERVER = "ihr-pop3-postfach"
USER = "ihr-benutzername"
PASSWORD = "ihr-passwort"
```

⁹<http://de.wikipedia.org/wiki/POP3>

```
puts "get mails from server #{POP3_SERVER} ..."
pop = Net::POP3.new(POP3_SERVER)
server = pop.start(USER, PASSWORD)
TaskMailReceiver.receive(server.mails[0].mail)
```

Der Aufruf führt ein Login am definierten POP3-Server *POP3_SERVER* durch und ruft die aktuellen Mails ab:

```
$ ruby test/getpop3.rb
```

Die erste Mail wird an die Methode *receive* der Klasse *TaskMailReceiver* übergeben. Das ist im Beispiel schon alles, Sie können nun Mails von einem POP3-Konto an die Anwendung übergeben. Probieren Sie es gleich aus. Lassen Sie das Programm *getpop3.rb* z.B. als Cron Job periodisch laufen.

Ausgehend von dem hier beschriebenen Prinzip, können Sie Ihre eigene Weiterleitung von Mails an Action Mailer bzw. Ihre Anwendung erstellen. Alternativ zum obigen Ansatz können Sie mit Hilfe des Skriptes *scripts/runner* eine Mail aus einer Datei an die Anwendung übergeben:

```
$ ruby script/runner TaskMailReceiver.receive(STDIN.read)
< email.txt
```

Eine weitere Alternative im Zusammenspiel mit einer *procmail*¹⁰-Konfiguration unter einem Unix-Derivat finden Sie im Internet auf den Rails-Seiten.¹¹

9.2.2 Empfang testen

Mit einem Test soll sichergestellt werden, dass die empfangene E-Mail auch korrekt von der Methode *receive* verarbeitet wird. Dazu schreiben wir einen Unit Test, der prüft, ob aus der übergebenen Mail ein Task erzeugt und in der Datenbank gespeichert wurde und ob die enthaltenen Daten den erwarteten entsprechen. Der Test sieht wie folgt aus:

Listing 9.28: test/unit/task_mail_receive_test.rb

```
class TaskMailReceiverTest < ActionMailer::TestCase
  tests TaskMailReceiver
  def test_receive
    assert_difference 'Task.count' do
      mail = TMail::Mail.new
      mail.subject = "Project Status: Rails Projekt"
      mail.to      = "chief@b-simple.de"
      ...
      TaskMailReceiver.receive(mail.encoded)
    end
    task = Task.find(:all, :order => :id).last
    assert_equal "Project Status: Rails Projekt", task.name
    ...
  end
end
```

¹⁰<http://www.procmail.org/>

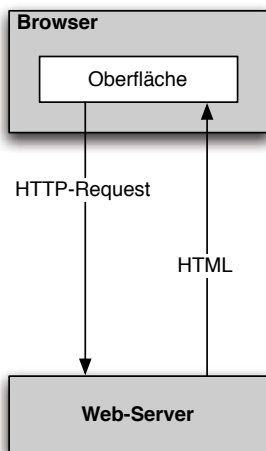
¹¹<http://wiki.rubyonrails.com/rails/show/HowToReceiveEmailsWithActionMailer>

Kapitel 10

Ajax

Klassische Web-Anwendungen arbeiten nach dem Request-/Response-Modell: Der Benutzer führt eine Interaktion im Browser durch, der Browser sendet einen Request an den Server, der Server bearbeitet den Request und liefert eine neue Seite an den Browser.

Klassische Web-Anwendung



Ajax Web-Anwendung

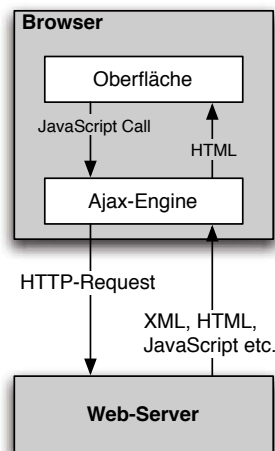


Abbildung 10.1: Klassische und Ajax-Web-Anwendung im Vergleich

Für die Zeit des Requests blockiert die Seite, und weiteres Arbeiten ist nicht möglich. Denken Sie an das Ausfüllen eines Formulars, bei dem nach dem Verlassen eines Feldes sofort eine Überprüfung der Eingabe erfolgen soll. Das ist bei Desktop-Anwendung gang und gäbe und komfortabler, als alle Daten einzugeben, die Seite abzuschicken und dann zu erfahren, dass in fünf Feldern falsche Eingaben erfolgten.

Das ständige Blockieren und Neuanzeigen der Seite nach einer Eingabe ist jedoch nicht praktikabel und benutzerfreundlich, und es wurde bisher darauf verzichtet.

Web-Anwendungen haben im Vergleich zu Desktop-Anwendungen eingeschränkte Möglichkeiten bezüglich komfortabler Oberflächen und Benutzerfreundlichkeit. Mit Ajax stehen neue Möglichkeiten zur Verfügung, die Web-Anwendungen sehr nahe an den Komfort von Desktop-Anwendungen heranzubringen, wenn nicht sogar mit ihnen gleichzuziehen. In diesem Kapitel werden wir einige Möglichkeiten kennenlernen.

Ajax steht für *Asynchronous JavaScript + XML* und ist eigentlich keine eigenständige Technologie, sondern eine Sammlung von Technologien wie DOM-Interaktion¹ und -Manipulation, XMLHttpRequest und JavaScript. Geprägt wurde der Begriff von Jesse James Garret, der im Februar 2005 einen Essay im Internet publizierte und darin die verwendeten Technologien unter dem Begriff Ajax zusammenfasste (vgl. [21]).

Zentrale Ajax-Komponente ist die *Ajax-Engine*, eine in allen modernen Browsern verfügbare Komponente für die asynchrone Kommunikation mit dem Webserver (siehe Abbildung 10.1). Die Ajax-Engine verarbeitet JavaScript-Aufrufe und erzeugt asynchrone HTTP-Requests an den Server. Clientseitig kehrt der Kontrollfluss unmittelbar nach Ausführung des JavaScript-Aufrufs an den Browser zurück, und der Benutzer kann weiterarbeiten, während der Server die Anfrage bearbeitet.

Während der Benutzer ganz normal weiterarbeitet, läuft im Browser eine zentrale Event-Loop, die neben den fortlaufenden Benutzerinteraktionen auch auf Antworten des Servers horcht. Der Server kann als Antwort auf den Request beliebige Daten senden: XML, HTML oder auch JavaScript.

Die Ajax-Engine nimmt die Antwort des Servers entgegen und fügt z.B. ein empfangenes HTML-Fragment an einer bestimmten Stelle in die geöffnete Seite ein oder führt ein empfangenes JavaScript im Browser aus. All dies passiert parallel zur normalen Arbeit des Benutzers. Teile der Oberfläche ändern sich dynamisch, ohne dass die Anwendung blockiert oder Seiten vollständig neu geladen werden müssen. Abbildung 10.2 stellt die klassische synchrone Kommunikation der Ajax-basierten asynchronen Kommunikation gegenüber und verdeutlicht so das wesentlich reaktionsfreudigere Antwortverhalten von Ajax-Anwendungen.

Die etwas dickeren schwarzen Balken markieren die Zeitspannen, die der Client nach dem Absetzen eines HTTP-Requests blockiert. Die Graphik macht deutlich, dass die Blockaden im Ajax-Modell der Anwendung wesentlich kürzer als in der klassischen Variante sind, obwohl die Request-Bearbeitungszeiten durch den Server in beiden Varianten identisch sind.

¹ DOM steht für *Document Object Model* und bezeichnet eine Programmierschnittstelle (API) für den Zugriff auf HTML- oder XML-Dokumente.

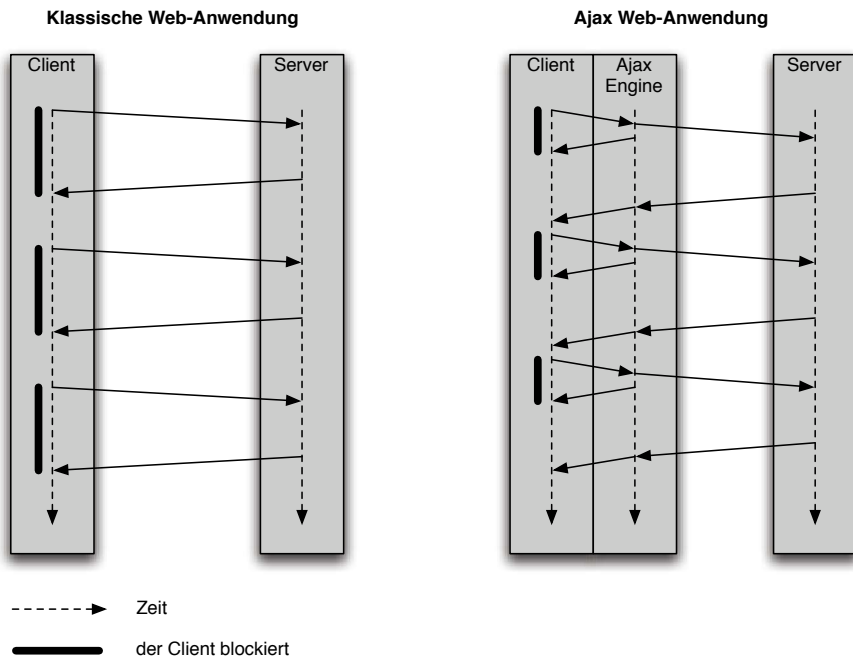


Abbildung 10.2: Synchron und asynchrone Kommunikation im Vergleich

10.1 Ajax und Rails

Rails ist eines der ersten Frameworks, das umfassende Ajax-Unterstützung anbietet. Es integriert dazu die JavaScript-Bibliotheken *Prototype* und *Scriptaculous* in jedes Projekt ([15], [16]).

Prototype (*prototype.js*) erweitert die Sprache JavaScript für die vereinfachte Entwicklung von dynamischen Web-Seiten. *Scriptaculous* (*controls.js*, *dragdrop.js* und *effects.js*) setzt auf *Prototype* auf und bietet Funktionen wie Drag and Drop, visuelle Effekte usw. Die JavaScript-Dateien beider Bibliotheken werden während der Erzeugung eines Rails-Projekts in das Verzeichnis *public/javascripts* kopiert.

Zusätzlich enthält ein Projekt die Datei *application.js*, die für Ihr eigenes JavaScript gedacht ist. Alle diese Dateien werden wie folgt importiert:

```
<%= javascript_include_tag :defaults %>
```

Haben Sie weitere, müssen Sie diese explizit einbinden:

```
<%= javascript_include_tag "mycode.js" %>
```

Rails stellt über Module wie *ActionView::Helpers::JavaScriptHelper* oder *ActionView::Helpers::PrototypeHelper* eine Reihe von Hilfsmethoden zum Aufruf der JavaScript-Funktionen zur Verfügung und erlaubt damit die Ajax-Programmierung in Ruby.

In den folgenden Abschnitten werden wir Beispiele einiger Hilfsmethoden zeigen. Im Abschnitt 10.8 beschreiben wir anschließend die RJS Templates, eine einfache und komfortable Möglichkeit, mehrere Bereiche einer Seite gleichzeitig zu aktualisieren. Unter <http://wiki.script.aculo.us/scriptaculous/show/Demos> finden Sie eine Reihe von Online-Beispielen zu den beschriebenen Funktionen.

10.2 Hello, Ajax World

Traditionell begrüßen Programmierer neue Technologien mit einem kleinen „Hello World“-Programm. Auch wir wollen uns dieser Tradition nicht entziehen und die Ajax-Welt mit einem freundlichen „Hallo“ begrüßen.

Wir schreiben uns dazu folgenden *DemoController*, der die Actions *index* und *hello_ajax_world* definiert:

Listing 10.1: app/controllers/demo_controller.rb

```
class DemoController < ApplicationController
  def index
  end

  def hello_ajax_world
    render :text => "AJAX World"
  end
end
```

Die Action *index* liefert den View *index*, der die JavaScript-Bibliotheken lädt und den Text „Hello“ und ein *div*-Element mit der ID *ajax_placeholder* definiert:

Listing 10.2: app/views/demo/index.html.erb

```
<%= javascript_include_tag :defaults %>

Hello,
<div id="ajax_placeholder">
!!! vorher !!!
</div>

<%= link_to_remote("Sag hallo",
                  :update => "ajax_placeholder",
                  :url => {:action => :hello_ajax_world}) %>
```

Spannend wird der View erst mit der Methode *link_to_remote*. Sie ist eine der Ajax-Hilfsmethoden und erzeugt einen Link mit dem Text „Sag hallo“. Die Methode sendet beim Anklicken des Links einen asynchronen Request an den Server.

Der *url*-Parameter gibt dabei die Controller-Action an, die für die Bearbeitung des Requests zuständig ist. Im Beispiel handelt es sich um die Action *hello_ajax_world*. Der *update*-Parameter spezifiziert den *div*-Container, dessen Inhalt durch die Antwort des Servers aktualisiert werden soll. Im Beispiel ist dies der *div*-Container mit der ID *ajax_placeholder*.

Der asynchrone Aufruf der Action *hello_ajax_world* liefert den Text „AJAX World“, der somit den Text „!!! vorher !!!“ ersetzt, und zwar ohne die gesamte Seite neu zu laden (siehe Abbildung 10.3).

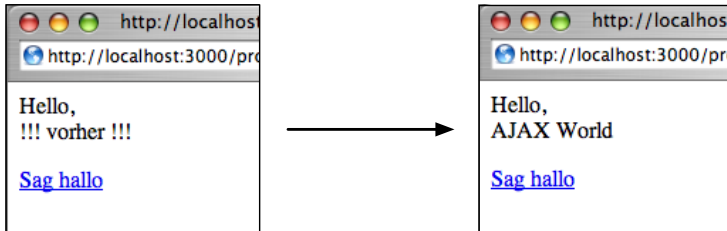


Abbildung 10.3: Schöne neue Ajax-Welt

10.3 Ajax-Formulare

Formulare und deren Inhalte werden durch Ausführung einer Submit-Action an den Server übertragen. Mit Hilfe des Ajax-Helpers *form_remote_tag* ist dies jetzt auch asynchron möglich. Der Benutzer kann Formulare abschicken und anschließend ganz normal weiterarbeiten, ohne zunächst auf die Antwort des Servers warten zu müssen. Das *form_remote_tag* arbeitet prinzipiell wie das bereits bekannte *form_tag*, d.h. es gibt definierte Start- und Ende-Tags, und die dazwischen enthaltenen Formularelemente werden beim Submit an den Server übertragen:

```
<%= form_remote_tag(:update => :list,
                    :url => {:action => :add} ) %>
  <%= text_field :task, :name %>
  <%= submit_tag 'Create' %>
</form>
```

Die *form_remote_tag*-Methode erwartet als Parameter eine Update-ID und eine URL. Die Update-ID gibt wie beim Remote-Link den HTML-Container vor, der beim Eintreffen der Server-Antwort aktualisiert wird. Der Parameter *:url* spezifiziert die beim Auslösen des Submits auszuführende Action. Neben dem Pflichtparameter *:action* akzeptiert der URL-Parameter beliebige weitere Parameter als Elemente der Hash, wie beispielsweise *:controller*.

10.3.1 Task-Schnellerfassung

Als Beispiel für die Anwendung von Ajax-Formularen soll die Taskerfassung von OnTrack vereinfacht werden. Zur Zeit ist es so, dass der Benutzer den *Add-Task*-Button drückt, woraufhin sich ein weiterer View zur Erfassung des neuen Tasks öffnet.

Für die vereinfachte Taskerfassung soll der Show-View für Iterationen um ein Textfeld zur Erfassung eines Tasknamens und um einen Button zum Erzeugen des neuen

Tasks erweitert werden (siehe Abbildung 10.4). Klickt der Benutzer auf den Button, erzeugt das System einen neuen Task mit dem im Textfeld enthaltenen Tasknamen. Die Taskerzeugung soll asynchron erfolgen. Der Benutzer kann also sehr schnell mehrere Tasks hintereinander erfassen, ohne auf die eigentliche Verarbeitung der Erfassung warten zu müssen.

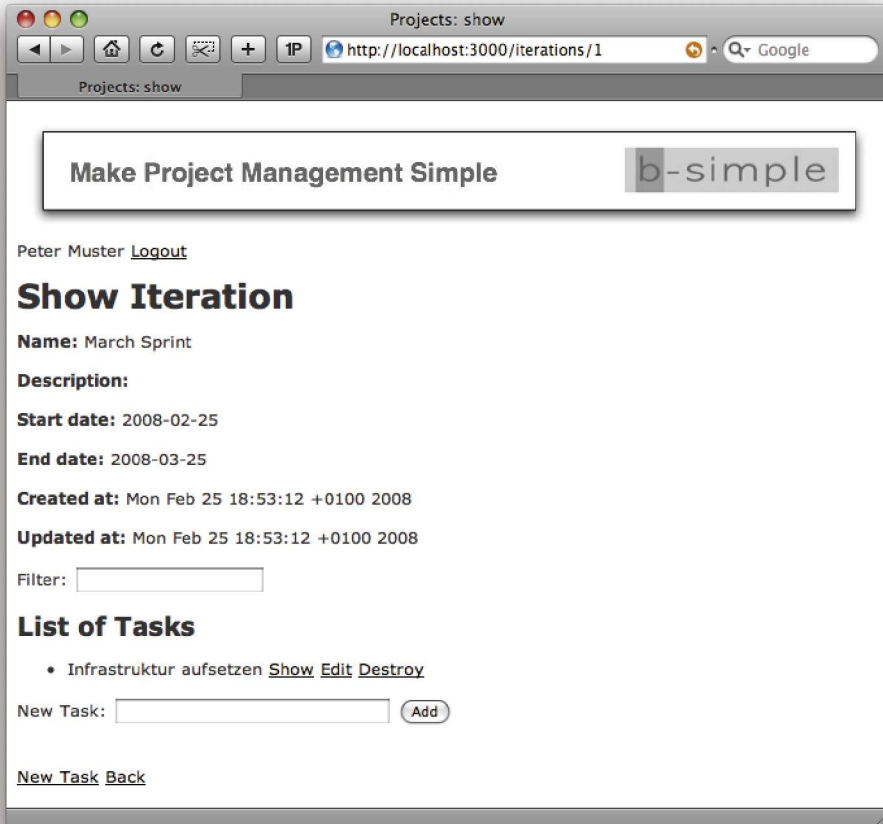


Abbildung 10.4: Komfortables Hinzufügen von Tasks dank Ajax

Als Erstes stellen wir sicher, dass alle JavaScript-Bibliotheken eingebunden sind:

Listing 10.3: app/views/application.html.erb

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

Danach überarbeiten wir den View *app/views/iterations/show.html.erb*, sodass die Taskliste beim Eintreffen einer Antwort für einen vorausgehenden Ajax-Add-Request um den neu erfassten Task erweitert wird. Bei der Anzeige der Taskliste gilt es jetzt also zwei Fälle zu unterscheiden:

1. Initiale Darstellung der Taskliste mit allen Tasks der Iteration
2. Erweiterung der Taskliste nach der Verarbeitung einer *add_task*-Action

Für die dynamische Erweiterung von Listen bietet Rails ein Lösungsmuster basierend auf Partial-Views an. Zunächst modifizieren wir den View *app/views/iterations/show.html.erb* so, dass die Tasks der Iteration vom Partial-View *app/views/projects/_task.html.erb* dargestellt werden:

Listing 10.4: *app/views/iterations/show.html.erb*

```
...
<div id="task_list">
  <ul id="list">
    <%= render :partial => 'tasks/task',
              :collection => @iteration.tasks %>
  </ul>
</div>
```

Der Partial-View *app/views/tasks/_task.html.erb* ist denkbar einfach. Wir definieren statt einer Tabelle eine Liste aller Tasks, da sich die Liste besser per JavaScript aktualisieren lässt. Pro Task sind entsprechende Links zum Anzeigen, Bearbeiten und Löschen angegeben. Wir verzichten der Einfachheit halber auf jegliches Design:

Listing 10.5: *app/views/tasks/_task.html.erb*

```
<li id="task_<%= task.id%>">
  <%= task.name %>
  <%= link_to 'Show', task_path(task) %>
  <%= link_to 'Edit', edit_task_path(task) %>
  <%= link_to 'Destroy', task_path(task),
            :method => :delete, :confirm => "Are you sure?" %>
</li>
```

Als Nächstes benötigen wir ein Textfeld für die Eingabe des Tasknamens sowie einen Submit-Schalter für die Erzeugung des neuen Tasks. Beide Elemente betten wir in ein Remote-Form des Show-Views ein, sodass die Task-Schnellerfassung asynchron via Ajax erfolgt:

Listing 10.6: *app/views/iterations/show.html.erb*

```
...
<%= form_remote_tag(:update => :list, :position => :bottom,
  :url => { :controller => :tasks, :action => :add_ajax_task,
          :iteration_id => @iteration },
  :html => { :id => 'add_task_form' } ) %>
  New Task: <%= text_field :task, :name %>
  <%= submit_tag 'Add' %>
</form>
```

Das *form_remote_tag* gibt als Update-Element die Taskliste mit der ID *list* an. Des Weiteren spezifizieren wir über den Parameter *:position* den Wert *bottom*, um den neuen Task an das Ende der Liste der existierenden Tasks anzufügen. Das Remote-Form referenziert die Action *add_ajax_task*, die als Parameter die ID der aktuell angezeigten Iteration übergeben bekommt. Die Action wird im *TasksController* implementiert:

Listing 10.7: app/controllers/tasks_controller.rb

```
def add_ajax_task
  iteration = Iteration.find(params[:iteration_id])
  @task = Task.new(params[:task])
  iteration.tasks << @task
  render :partial => 'task'
end
```

Es handelt sich um eine ganz normale Action, die eine Iterations-Instanz lädt und ihr ein neues Task-Objekt hinzufügt. Abschließend liefert die Action den Partial-View *task* und übergibt dem View das neu erzeugte Task-Objekt. Das war schon alles. Testen Sie gleich einmal die Erzeugung neuer Tasks per Ajax-Helfer.

10.3.2 Ajax-Formulare und JavaScript

Im Beispiel aus dem vorigen Abschnitt wird der Formularinhalt mit Hilfe eines Submit-Buttons an den Server übertragen. Für einige Anwendungsfälle ist jedoch die Ausführung eines JavaScripts anstelle des direkten Submits sinnvoller. Die Ausführung des Scripts bietet die Möglichkeit, vor oder nach dem Submit zusätzliche Funktionen auszuführen, was bei einem direkten Submit nicht möglich wäre.

Für die Ausführung von JavaScript muss der View-Helper *form_remote_tag* um einen HTML-Parameter erweitert werden, der die ID des Formulars angibt:

```
<%= form_remote_tag(...,
  :url => { ... },
  :html => { :id => 'myformid' } ) %>
```

Die ID des Formulars kann jetzt in JavaScript-Funktionen für die Ausführung eines Submits verwendet werden:

```
<a href="#" onclick="$('myformid').onsubmit();">Add Task</a>
```

10.4 Feldbeobachter

Ein weiterer Ajax-Wrapper ist die Funktion *observe_field*. Die Funktion prüft den Inhalt eines Formularelements in regelmäßigen Abständen und meldet Änderungen automatisch an den Server. Die Funktion bietet weitere Optionen für die Gestaltung sehr interaktiver und schnell reagierender Web-Seiten.

Ein Beispiel dafür ist eine dynamische Suchliste. Hierbei gibt der Benutzer einen Suchbegriff in ein dafür vorgesehenes Feld ein, und das System reagiert auf die

Eingabe jedes weiteren Buchstabens mit einer Einschränkung der angezeigten Ergebnismenge. Eingabe und Suche finden quasi parallel statt, ohne dass der Benutzer zunächst den Suchbegriff eingeben und anschließend einen Schalter zum Starten der Suche aktivieren muss.

Die *observe_field*-Methode besitzt folgende Syntax:

```
<%= observe_field(:search_field,
                  :frequency => 0.2,
                  :update => "result_list",
                  :url => {:action => "do_the_search",
                          :controller => "search"}) %>
```

Der erste Parameter gibt die DOM-ID des zu beobachtenden Feldes an (im Beispiel *:search_field*). Der zweite Parameter ist eine Hash mit folgenden Optionen:

:frequency

Frequenz in Sekunden, mit der das Feld untersucht wird.

:update

DOM-ID des HTML-Containers, dessen Inhalt mit der HTML-Antwort der aufgerufenen Action ersetzt werden soll.

:url

URL (:action, :controller, eventuell optionale Parameter) der bei Änderung von Feldinhalten zu aktivierenden Controller-Action.

:with

Die Methode *observe_field* überträgt den Inhalt des Felds als „raw“ POST-Data an den Server, und die aufgerufene Action liest den Wert über die *raw_post*-Methode des Requests aus, z.B.:

```
def filter
  task_name = '%' + request.raw_post + '%'
  ...
end
```

Durch die Angabe von *:with => xxx* kann der Inhalt aus dem Request-Parameter über *params[xxx]* ausgelesen werden. Dies ist notwendig, wenn Sie Zugriff auf mehr als ein Feld des aktuell angezeigten Formulars benötigen. Angenommen, ein View enthält einen Filterbereich, der aus mehr als einem Eingabefeld besteht:

```
<%= form_tag ...
  <%= text_field_tag :taskname %>
  <%= text_field_tag :taskprio %>
  ...
```

Immer wenn sich der Wert eines der Felder ändert, soll die zugehörige Filter-Action aufgerufen werden. Die Filter-Action benötigt für einen korrekten Filtermechanismus nicht nur den Wert des geänderten Felds, sondern die Werte aller den Filter definierenden Felder. Der folgende *observe_field*-Aufruf gewährleistet dieses Verhalten:


```
<%= observe_field(:taskname,
  :frequency => 0.2,
  :url => {:action => :do_the_search},
  :with => "'taskname=' + $('taskname').value + '&' +
    'taskprio=' + $('taskprio').value") %>
```

Der *with*-Parameter beschreibt die an den Server zu übermittelnden Werte als Paa-re, bestehend aus dem Namen eines Felds (z.B. *taskname*) und dem Wert des Felds (z.B. *\$('taskname').value*). Feldname und Wert sind durch ein Gleichheitszeichen ge-trennt. Mehrere Parameter werden durch das Zeichen & getrennt. Die von *observe_field* aufgerufene Action *do_the_search* erhält die im Parameter *with* angegebenen Formularwerte als normale Request-Parameter im *params*-Hash:

```
def do_the_search
  task_name = params[:taskname]
  task_prio = params[:taskprio]
  ...
end
```

Damit die Filterfunktion auf alle Felder des Filters reagiert, wird ein entsprechender *observe_field*-Aufruf für jedes Feld des Filterbereichs benötigt. Oder Sie verwenden *observe_form* für die gesamte Form.

10.4.1 Beispiel: Tasks filtern

Unser Projektmanagement-System OnTrack bietet verschiedene Anwendungsmöglichkeiten für den Einsatz von Feldbeobachtern. Eine ist z.B. die dynamische Einschränkung der Tasks einer Iteration in Abhängigkeit eines eingestellten Filters (siehe Abbildung 10.5).

Der Show-View einer Iteration muss dafür um ein Filter-Textfeld erweitert werden. Ändert der Benutzer dieses Feld, schränkt das System automatisch die angezeigte Taskliste auf die dem Filter entsprechenden Tasks ein. Zunächst die Erweiterung des Show-Views:

Listing 10.8: app/views/iterations/show.html.erb

```
...
Filter: <%= text_field_tag :filter %>
<%= observe_field(:filter, :frequency => 0.2,
  :with => :filter_text, :update => :task_list,
  :url => {:controller => :tasks,
    :action => :filter,
    :iteration_id => @iteration}) %>
...
```

Das neue Textfeld *:filter* dient der Eingabe des Suchbegriffs. Die Methode *observe_field* sorgt dafür, dass das Filterfeld alle 0,2 Sekunden auf Veränderungen geprüft wird. Ist dies der Fall, erfolgt ein asynchroner Aufruf der Action *filter* des *TasksControllers*. Der Action wird als zusätzlicher Parameter die ID der aktuell angezeigten Iteration übergeben. Der folgende Codeausschnitt zeigt die *filter*-Action:

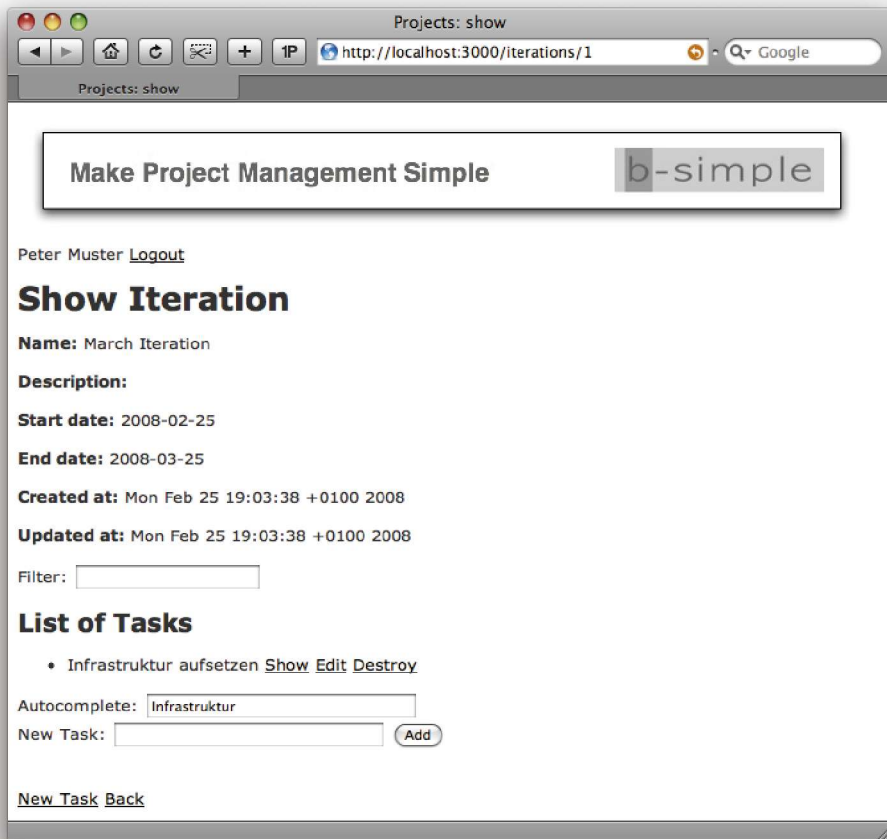


Abbildung 10.5: Eine Filterfunktion für Tasks

Listing 10.9: app/controllers/tasks_controller.rb

```
def filter
  @tasks = Task.find(:all,
    :conditions => ['name like ? and iteration_id = ?',
      "%#{params[:filter_text]}%", params[:iteration_id]])
  render :partial => 'task_list', :object => @tasks
end
```

Der Inhalt des Filterfelds und die ID der aktuell angezeigten Iteration dienen als Eingabeparameter für den Finder-Aufruf, der eine Liste der den Filterkriterien genügenden Tasks ermittelt. Anschließend liefert die Action den neu zu programmierenden Partial-View `_task_list.html.erb`, der die gefilterte Taskliste als `:object`-Parameter erhält:

Listing 10.10: app/views/tasks/_task_list.html.erb

```
<ul id="list">
```

```
<%= render :partial => 'task', :collection => @tasks %>
</ul>
```

Der View stellt die gefilterte Taskliste dar, wobei die einzelnen Task-Objekte von dem bereits existierenden Partial-View *_task.html.erb* ausgegeben werden (siehe Listing 10.5).

10.5 Callback-Methoden

In asynchron arbeitenden Anwendungen kommen den Callback-Methoden eine zentrale Rolle zu, da der Anwendung durch Aufruf dieser Methoden die Beendigung bestimmter Aktionen signalisiert werden kann. Callbacks können von den beiden Methoden *link_to_remote* und *form_remote_tag* sowie von den *observe_*-Methoden verwendet werden, indem über zusätzliche Parameter zu Beginn bzw. am Ende einer bestimmten Aktion eine JavaScript-Methode aufgerufen wird. Folgende Callback-Parameter existieren:

:before

Wird aufgerufen, bevor der Aufruf initialisiert wird.

:after

Wird aufgerufen, nachdem der Aufruf initialisiert wurde, und vor *:loading*.

:loading

Wird beim Start des Requests aufgerufen.

:loaded

Wird nach Beendigung des Requests aufgerufen, d.h. nachdem die Response beim Browser angekommen ist.

:interactive

Wird aufgerufen, wenn der Benutzer mit den geladenen Daten arbeiten kann, auch wenn sie noch nicht vollständig zum Browser übertragen wurden.

:success

Wird aufgerufen, wenn der Ajax-Aufruf vollständig abgeschlossen ist, und kein Fehler auftrat (Rückgabewert gleich 2XX).

:failure

Wird aufgerufen, wenn der Ajax-Aufruf vollständig abgeschlossen ist und ein Fehler auftrat (Rückgabewert ungleich 2XX).

:complete

Wird aufgerufen, wenn der Ajax-Aufruf vollständig abgeschlossen ist und nach *:success* oder *:failure*.

Das folgende Beispiel zeigt einen Ajax-Link, der beim Absetzen des Aufrufs die JavaScript-Funktion *ajax_started* und beim Eintreffen der zugehörigen Antwort die Funktion *ajax_finished* aufruft:

```
<%= link_to_remote("Sag hallo",  
  :update => "ajax_placeholder",  
  :loading => "ajax_started()",  
  :complete => "ajax_finished()",  
  :url => {:action => :hello_ajax}) %>
```

10.6 Drag and Drop

Kaum eine Rich-Client-Anwendung kommt heutzutage ohne „Drag and Drop“ aus. Warum sollte das in Web-Anwendungen anders sein? Haben Sie sich nicht auch schon einmal gewünscht, die Bücher in einem Online-Buchladen einfach mit der Maus zu greifen und in den Einkaufswagen zu ziehen?

Rails unterscheidet zwischen *draggable*- und *drop receiving*-Elementen. Draggable-Elemente können mit der Maus hochgenommen und über die Seite bewegt werden. Ein Objekt wird durch Aufruf der Methode *draggable_element* zu einem draggable-Objekt. Die Methode erwartet als Parameter die DOM-ID des Objekts:

```
<%= draggable_element :drag, :revert => true %>
```

Der Parameter *:revert* ist optional und gibt an, ob die Drag-Operation ein Verschieben oder ein Kopieren ist. Erhält der Parameter den Wert *true*, wird das Objekt kopiert, d.h. Quelle und Ziel des Vorgangs enthalten nach dem Loslassen eine visuelle Kopie des bewegten Objekts (hier Bücher). Dieses Verhalten wird in Abbildung 10.6 deutlich, in der ein in den Einkaufswagen bewegtes Objekt weiterhin in der Liste verfügbarer Bücher angezeigt bleibt. Wäre der *:revert*-Parameter in diesem Beispiel *false*, verschwände das Buch nach dem Verschieben in den Einkaufswagen aus der oberen Liste.

Drop receiving-Elemente markieren Ziele für draggable-Elemente. Der Aufruf der Methode *drop_receiving_element* macht ein DOM-Objekt zu einem solchen Ziel-element. Neben der DOM-ID des Objekts erwartet die Methode die URL einer Controller-Action, die beim Fallenlassen eines draggable-Objekts aufgerufen wird:

```
<%= drop_receiving_element :drop,  
  :url => {:controller => "shop", :action => "drop" } %>
```

10.6.1 Einkaufen per Drag and Drop

Wir wollen Ihnen die „Drag and Drop“-Funktionalität am Beispiel eines Online-Buchladens demonstrieren. Aus einer Liste von Büchern soll der Benutzer mit der Maus ein Buch nehmen, über einen Einkaufswagen ziehen und dort fallen lassen (siehe Abbildung 10.6).

Für das Beispiel erzeugen wir zunächst ein Modell *Book*:

```
$ ruby script/generate model book title:string
```

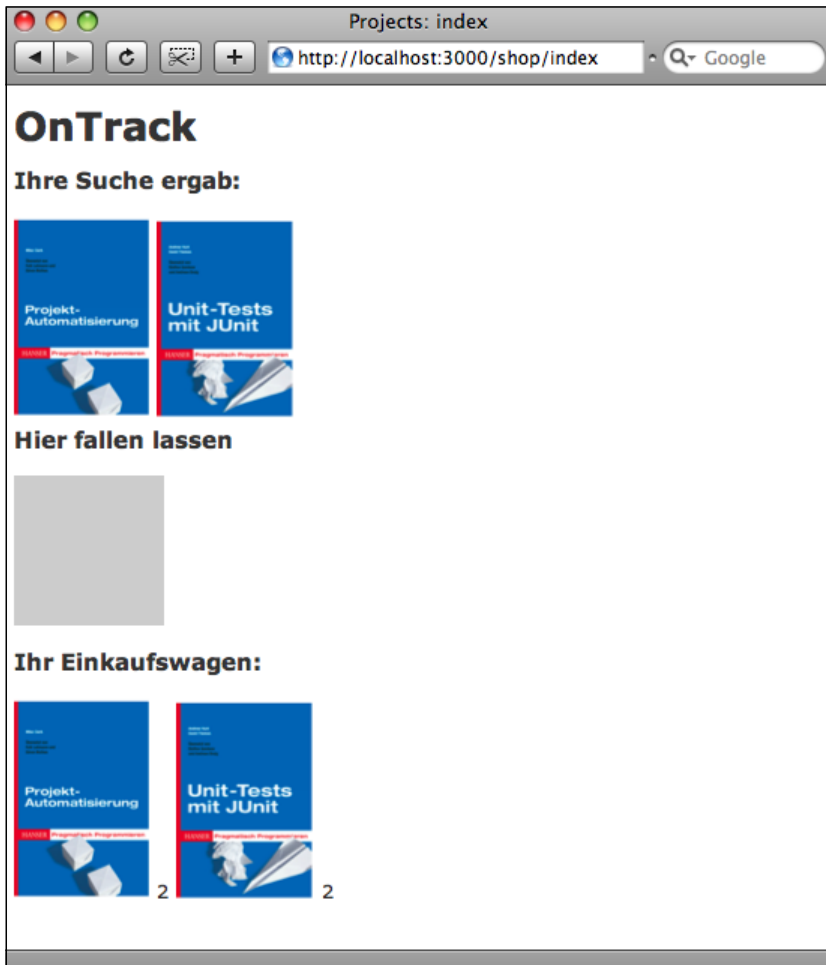


Abbildung 10.6: Schöner einkaufen dank Drag and Drop

Listing 10.11: db/migrate/create_books.rb

```
class CreateBooks < ActiveRecord::Migration
  def self.up
    ...
    Book.create(:title => "Projekt-Automatisierung")
    Book.create(:title => "Unit-Tests mit JUnit")
  end
  ...
end
```

Das Migrationsskript erzeugt zwei Bücher und wird wie bekannt ausgeführt:

```
$ rake db:migrate
```

Anschließend generieren wir den Controller:

```
$ ruby script/generate controller shop
```

Der Einfachheit halber definieren wir die Action *index* wie folgt. Sie lädt die Liste aller Bücher und setzt den Einkaufskorb in der Session zurück. Das ist nicht besonders realistisch, soll aber für das Beispiel reichen:

Listing 10.12: `app/controllers/shop_controller.erb`

```
class ShopController < ApplicationController
  def index
    @books = Book.find(:all)
    session[:cart] = {}
  end
end
```

Die Action liefert den View *index*, der wie folgt aussieht:

Listing 10.13: `app/views/shop/index.html.erb`

```
<h3>Ihre Suche ergab:</h3>
<div style="margin-bottom:20px;height:120px;">
  <% for book in @books %>
    <%= image_tag "/images/books/book_#{book.id}.png",
      :id => "book_#{book.id}",
      :alt => book.title,
      :class => "books" %>
    <%= draggable_element "book_#{book.id}", :revert => true %>
  <% end %>
</div>

<h3>Hier fallen lassen</h3>
<div id="drop"
  style="height:100px; width:100px; background-color:#CCC"></div>

<h3>Ihr Einkaufswagen:</h3>
<div id="cart"><%= render :partial => "cart" %></div>

<%= drop_receiving_element :drop,
  :update => "cart", :url => { :action => "add" },
  :accept => "books" %>
```

Es wird über eine Liste der gefundenen Bücher *@books* iteriert und für jedes enthaltene Buch ein kleines Bild dargestellt (*image_tag*). Jedes dieser Bilder wird zu einem draggable Element deklariert, indem der Methode *draggable_element* die ID des erzeugten Bilds übergeben wird.

Im Anschluss an die Bücherliste folgt der Drop-Bereich als grauer Kasten. Hier lassen Sie das Buch-Bild fallen, um es in den Einkaufskorb zu legen. Der Bereich ist als div-Container mit der ID *drop* definiert. Diese ID wird auch in der Methode *drop_receiving_element* weiter unten übergeben, damit diese auf ein Fallenlassen innerhalb dieses div-Containers reagiert.

Danach folgt der Einkaufskorb, der innerhalb des div-Containers mit der Id *cart* angezeigt wird. Wir trennen im Beispiel den Drop-Bereich und den Einkaufswagen,

was nicht zwingend notwendig wäre. Sie müssen nur sicherstellen, dass der Drop-Bereich für den Benutzer immer ersichtlich und sichtbar ist.

Die Methode *drop_receiving_element* am Ende des Views reagiert auf das Fallenlassen eines Bildes im div-Container *drop* und ruft die Action *add* auf. Das Ergebnis (der neue Warenkorb) wird in den div-Container *cart* eingefügt (Option *:update*).

Die Option *:accept* sorgt dafür, dass die Methode nur auf Objekte von der Klasse *books* reagiert. Wir haben dazu im Drag-Bereich jedes Bild mit diesem Wert versehen (*:class => books*). Auf diese Weise können Sie die Menge von akzeptierten Objekten einschränken.

Der folgende Codeausschnitt zeigt die *add*-Action des *ShopControllers*:

Listing 10.14: *app/controllers/shop_controller.rb*

```
def add
  book_id = params[:id].split("_")[1]
  session[:cart][book_id] =
    (session[:cart][book_id] || 0) + 1
  render :partial => 'cart'
end
```

Der Einkaufswagen ist in der Session unter dem Schlüssel *:cart* gespeichert. Das Cart-Objekt ist eine Hash, die als Schlüssel die ID des Buches und als Wert die Anzahl der gewünschten Bücher enthält. Die *add*-Methode inkrementiert diesen Wert und liefert anschließend den Partial-View *_cart*:

Listing 10.15: *app/views/shop/_cart.html.erb*

```
<% unless session[:cart].blank? %>
  <% session[:cart].each do |book,quantity| %>
    <%= image_tag "/images/books/book_#{book}.png",
      :class => "cart-items", :id => "item_#{book}" %>
    <%= quantity %>
  <% end %>
<% end %>
```

Der View iteriert über die im Cart-Objekt enthaltenen Bücher und stellt für jedes Buch die zugehörige Graphik, gefolgt von der Anzahl, dar. Hipp, oder?

10.6.2 Sortieren per Drag and Drop

Eine andere Verwendungsart für „Drag and Drop“ ist das Sortieren von Listen. Hierfür stellt Rails die Methode *sortable_element* zur Verfügung:

```
<%= sortable_element :list, :url => { :action => "order" } %>
```

Der Parameter *list* gibt die Liste an, deren Elemente sortiert werden sollen. Die Hash *:url* spezifiziert die Action, die beim Loslassen eines zuvor aufgenommenen und bewegten Elements asynchron aufgerufen wird. Zusätzlich akzeptiert *sortable_element* den optionalen Parameter *:update*, der den beim Fallenlassen des Objekts zu aktualisierenden HTML-Container spezifiziert.

Wir wollen die Sortierfunktionalität am Beispiel der OnTrack-Taskliste demonstrieren. Der Benutzer soll die Möglichkeit haben, einen Task mit der Maus zu greifen und an beliebig anderer Stelle in die Liste wieder einzufügen (siehe Abbildung 10.7).

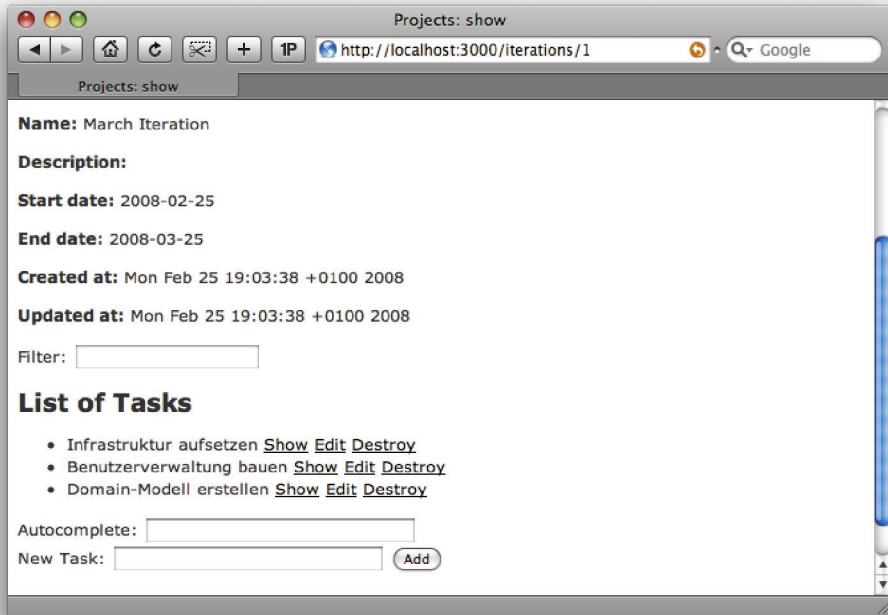


Abbildung 10.7: Das Sortieren von Tasks wird einfacher.

Zunächst erweitern wir den Show-View für Iterationen um einen Aufruf der Funktion *sortable_element*, der die Elemente der Taskliste *list* sortierbar macht:

Listing 10.16: app/views/iterations/show.html.erb

```
...
<div id="task_list">
  <ul id="list">
    <%= render :partial => 'task',
              :collection => @iteration.tasks %>
  </ul>
</div>

<%= sortable_element :list,
  :url => { :controller => :tasks, :action => 'order_tasks' } %>
```

Der *sortable_element*-Aufruf spezifiziert als Drop-URL die Action *order_tasks* des *TasksController*. Diese Action wird immer dann aufgerufen, wenn der Benutzer einen Task an einer bestimmten Stelle fallen lässt.

Die Erweiterung des Views ist so weit ausreichend, dass schon jetzt die Drag and Drop-Funktionalität zur Verfügung steht. Sie können einen Task mit der Maus greifen, ihn an eine andere Position ziehen und dort fallen lassen. Allerdings verändert sich die Position des Tasks nur oberflächlich. Ein Reload im Browser stellt die alte Sortierung wieder her.

Damit die neue Ordnung von Dauer bleibt, muss die aktuelle Reihenfolge der Tasks auch in der Datenbank gespeichert werden. Und genau diesem Zweck dient die im URL-Parameter von *sortable_element* angegebene Action *order_tasks*.

Damit die vom Benutzer per Drag and Drop vorgegebenen Task-Reihenfolgen berücksichtigt werden, sind zunächst einige Änderungen am zugrunde liegenden Domain-Modell erforderlich. Das Thema *Reihenfolgen in 1:N-Beziehungen* haben wir bereits in Abschnitt 4.13.1 ausführlich behandelt, sodass wir an dieser Stelle die notwendigen Schritte nur kurz wiederholen:

- Erweiterung der Tabelle *tasks* um ein Feld *position* vom Typ *Integer*
- Erweiterung der Klasse *Task* um einen Aufruf der Methode *acts_as_list*
- Erweiterung der *has_many-Deklaration* von *Iteration* um den Parameter *:order*

Für die Schema-Änderung erzeugen wir ein neues Migrationsskript (vgl. Abschnitt 3.5.2):

```
$ ruby script/generate migration add_position
exists db/migrate
create db/migrate/007_add_position.rb
```

Dieses sieht wie folgt aus:

```
class AddPosition < ActiveRecord::Migration
  def self.up
    add_column :tasks, :position, :integer
  end
  def self.down
    remove_column :tasks, :position
  end
end
```

Die Aktualisierung der Datenbank erfolgt wie gehabt durch:

```
$ rake db:migrate
```

Die erforderlichen Änderungen in den Modellen sind im folgenden Codeausschnitt zusammengefasst. Die Angabe der Option *:scope* sorgt dafür, dass nur die Tasks zu einer Iteration sortiert werden und nicht alle:

```
class Task < ActiveRecord::Base
  acts_as_list :scope => :iteration_id
  ...
end
```

```
class Iteration < ActiveRecord::Base
  has_many :tasks, :order => :position
  ...
end
```

Denken Sie daran, dass das Plugin *acts_as_list* installiert sein muss (vgl. 4.13.1). Nach der Anpassung des Domain-Modells können wir uns an die Umsetzung der *order*-Action machen:

Listing 10.17: `app/controllers/tasks_controller.rb`

```
def order_tasks
  pos = 1
  params[:list].each do |task_id|
    Task.find(task_id).update_attribute(:position, pos)
    pos += 1
  end
  render :nothing => true
end
```

Die *params*-Hash enthält unter dem Schlüssel *:list* (dies ist die ID der Task-Liste im Show-View) ein Array mit den ID-Werten der Tasks der bearbeiteten Iteration. Die Reihenfolge der IDs entspricht der visuellen Reihenfolge der Tasks im View. Es ist somit ausreichend, sequentiell über die Liste zu iterieren und das *position*-Attribut der Task-Objekte entsprechend zu aktualisieren. Da der View bereits vor Ausführung der Action auf dem neuesten Stand war, liefert die Action „nichts“ zurück, d.h. es wird kein Element auf der den Request auslösenden HTML-Seite aktualisiert.

10.7 Autocompletion

Autocompletion ist eine Funktion, die den Benutzer beim Ausfüllen von Formularfeldern unterstützt, indem das System eine Liste von potenziellen Eingabewerten vorschlägt.

Für die Verwendung müssen Sie zunächst das Plugin *auto_complete* installieren:

```
$ script/plugin install auto_complete
```

Im Standardfall bezieht sich ein Autocomplete-Feld immer auf ein bestimmtes Attribut einer Modellklasse. Deklariert wird das Feld im View mit Hilfe des Formular-Helpers *text_field_with_auto_complete*:

```
<%= text_field_with_auto_complete :task, :name %>
```

Der Codeausschnitt führt die automatische Vervollständigung für das Attribut *name* der Modellklasse *Task* durch. Immer wenn der Benutzer den Inhalt des Felds verändert, liest das System eine Liste von Tasks aus der Datenbank, die den bereits eingegebenen Teilstring im Namen enthalten. Diese Liste wird dem Benutzer als Vorschlagsliste präsentiert, aus der er einen Eintrag auswählen und in das Feld übernehmen kann. Intern setzt Rails bei jeder Änderung des Feldinhalts einen Ajax-Aufruf ab, der als Ergebnis die entsprechend gefilterte Liste liefert.

Damit der Server auf den Ajax-Aufruf reagieren kann, muss der zugehörige *Tasks-Controller* die Action *auto_complete_for_task_name* definieren, die die entsprechenden Datenbankaufrufe enthält. Der View *show.html.erb* wird aber vom *Iterations-Controller* ausgeliefert, und die Methode ruft daher die Action auf diesem Controller auf. Im Beispiel müssen wir den Controller und die Action daher explizit angeben. Dadurch verlieren wir etwas von der knappen und eleganten Art des Aufrufs, aber in diesem Fall geht es nicht anders:

```
<%= text_field_with_auto_complete :task, :name, {},
      :url => { :controller => :tasks,
               :action => :auto_complete_for_task_name } %>
```

Im Controller stellt uns Rails über die Methode *auto_complete_for* eine Defaultimplementierung der Action *auto_complete_for_task_name* bereit:

```
class TasksController < ApplicationController
  auto_complete_for :task, :name
end
```

Wenn Ihnen das nicht reicht, definieren Sie die Methode selbst und passen das Verhalten individuell an:

```
class TasksController < ApplicationController
  def auto_complete_for_task_name
    value = params[:task][:name]
    @tasks = Task.find(:all,
      :conditions => ['lower(name) like ?',
                    value.downcase + '%'])
    render :partial => 'task_list'
  end
end
```

Das alleine reicht aber noch nicht, da Rails bei einem POST den Sicherheitsfilter *verify_authenticity_token* vor der Action ausführt (vgl. 6.3.2). Da *text_field_with_auto_complete* keinen Token (*authenticity_token*) im Request mitliefert, wird der Filter den Aufruf der Action *auto_complete_for_task_name* verhindern. Wir schalten in diesem Fall daher die Prüfung über *skip_before_filter* ab:

```
class TasksController < ApplicationController
  skip_before_filter :verify_authenticity_token,
    :only => :auto_complete_for_task_name
  auto_complete_for :task, :name
end
```

Jetzt steht der Verwendung der Autocompletion nichts mehr im Wege.

10.8 RJS Templates

Im Gegensatz zu den herkömmlichen Templates, die HTML oder XML generieren, erzeugen Remote JavaScript Templates (kurz RJS) JavaScript-Code, der nach

der Auslieferung im Browser ausgeführt wird. Dies ermöglicht, mehrere Bereiche der Seite gleichzeitig zu aktualisieren, ohne die Seite neu laden zu müssen. Die Templates werden dabei in Ruby programmiert, woraus Rails den entsprechenden JavaScript-Code erzeugt. Die Entwicklung von RJS ist damit ohne große JavaScript-Kenntnisse möglich.

10.8.1 Hinzufügen von Tasks visuell optimieren

Um uns mit RJS Templates vertraut zu machen, werden wir im ersten Schritt das Hinzufügen von Tasks zu einer Iteration etwas aufpeppen. Der gerade hinzugefügte Task soll kurzfristig durch einen gelben Hintergrund hervorgehoben werden.

Wir passen hierzu zunächst den Aufruf an den Server an, indem wir die Optionen `:update` und `:position` im View löschen:

Listing 10.18: `app/views/iterations/show.html.erb`

```
...
<%= form_remote_tag(
  :url => { :controller => :tasks, :action => :add_ajax_task,
           :iteration_id => @iteration },
  :html => { :id => 'add_task_form' } ) %>
  New Task: <%= text_field :task, :name %>
  <%= submit_tag 'Add' %>
</form>
```

Die Option `:update` erzeugt nämlich JavaScript-Code, der nur das eine DOM-Objekt aktualisiert, das über `:update` definiert ist bzw. war (im Beispiel `list`). Dabei wird HTML-Code zurückgeliefert und an die Stelle des DOM-Objekts eingefügt (vgl. 10.2).

Da wir nun mehrere DOM-Objekte aktualisieren wollen, benötigen wir vom Server stattdessen JavaScript-Code, der selbst alle zu aktualisierenden Objekte definiert und im Browser ausgeführt wird. Der Code kann darüber hinaus weitere visuelle Effekte enthalten.

Hinweis: Sollten Sie einmal statt des erwarteten HTML-Codes JavaScript-Code in Ihrem View sehen, so haben Sie vermutlich den `:update`-Parameter nicht gelöscht oder keine Ajax-Methode verwendet (z.B. `form_tag` statt `form_remote_tag`).

Außerdem fügen wir über den Tag `:html` die DOM-ID `add_task_form` hinzu, um die Form später eindeutig referenzieren zu können (vgl. Listing 10.20).

Die aufgerufene Action `add_ajax_task` wird nun so geändert, dass sie statt HTML- nun JavaScript-Code liefert. Dazu ersetzen wir den Aufruf von `render` wie folgt:

Listing 10.19: `app/controllers/tasks_controller.rb`

```
def add_ajax_task
  iteration = Iteration.find(params[:id])
  @task = Task.new(:name => params['task']['name'])
  iteration.tasks << @task
  render :template => 'tasks/add_ajax_task'
end
```

Bei Ausführung der Action wird nun aus dem gleichnamigen RJS Template JavaScript-Code generiert und ausgeliefert. Dabei wird das RJS Template im Gegensatz zu den RHTML-Templates nicht als Partial View gerendert. Rails ist klar, dass RJS Templates immer nur Teilbereiche der Seite per JavaScript aktualisieren und nicht die gesamte Seite neu ausgeliefert wird.

Rails sucht im entsprechenden Verzeichnis nach einem Template *add_ajax.task* mit der Endung *.erb*, *.builder* und *.rjs*, und zwar in der angegebenen Reihenfolge. Stellen Sie in diesem Fall also sicher, dass sich kein Template mit gleichem Namen und Endung *.html.erb* oder *.builder* im Template-Verzeichnis befindet. Eine weitere Möglichkeit ist die Verwendung der Methode *respond_to*, die in Abschnitt 5.5 beschrieben ist.

Es folgt im nächsten Schritt die Erstellung des RJS Templates:

Listing 10.20: `app/views/tasks/add_ajax.task.rjs`

```
# Füge den neuen Task (per Partial View) am Ende der Liste ein.
page.insert_html :bottom, 'list', :partial => 'task'

# Hebe den neuen Task kurzfristig hervor
page.visual_effect :highlight, "task_#{@task.id}", :duration => 3.0

# Setze die Form für eine neue Eingabe zurück
page.form.reset 'add_task_form'

# Aktualisiere Liste, weil andernfalls der hinzugefügte Task
# sich nicht per Drag and Drop sortieren lässt.
page.sortable 'list', :url => { :action => "order_task" }
```

Bei einem RJS Template handelt es sich im Prinzip um eine Reihe von Definitionen zur Aktualisierung von Teilbereichen der Seite. Diese Definitionen werden in Ruby geschrieben und von Rails in entsprechenden JavaScript-Code umgesetzt. Dazu steht in jedem Template automatisch das vordefinierte Objekt *page* der Klasse *JavaScriptGenerator*² zur Verfügung. Die Klasse kapselt den JavaScript-Code durch einfache Schnittstellen und ermöglicht so die Verwendung von RJS Templates ohne das explizite Programmieren von JavaScript. Der erzeugte JavaScript-Code wird letztlich durch die Action (im Beispiel *add_ajax.task*) an den Browser zurückgeliefert und in diesem ausgeführt.

Genau genommen erfolgt die Verarbeitung des Templates innerhalb eines Blocks zur Methode *PrototypeHelper#update_page*. Diese erzeugt die *page*-Instanz und übergibt sie dem Template:

```
update_page do |page|
  # Ausführung des Templates
  ...
end
```

Im Falle der Templates wird uns die Angabe von *update_page* durch Rails abgenommen.

² `ActionView::Helpers::PrototypeHelper::JavaScriptGenerator`

Ein Blick in die API-Dokumentation zur Klasse *JavaScriptGenerator* zeigt eine Reihe von Methoden, die für die Aktualisierung der Seite zur Verfügung stehen. Im Listing 10.20 aktualisieren wir über die Methode *insert_html* die Task-Liste, indem wir durch die Option *:bottom* den neuen Task am Ende der Liste einfügen. Der hierfür notwendige HTML-Code wird über den Aufruf des Partial Views *task* geliefert (vgl. Listing 10.5).

Würden wir an dieser Stelle aufhören, hätten wir das gleiche Ergebnis wie ohne RJS Templates: Die Liste der Tasks wird am Ende um den neuen Task erweitert, ohne die Seite neu laden zu müssen. Im Grunde ist der Aufruf des Partial Views *task* aus Listing 10.7 nur ins Template verschoben worden.

Wir gehen nun einen Schritt weiter und heben den neuen Task kurzzeitig durch einen gelben Hintergrund hervor. Dazu dient uns die Methode *visual_effect*, mit deren Hilfe visuelle Effekte im Browser erzielt werden können. Eine Übersicht finden Sie unter <http://wiki.script.aculo.us/scriptaculous>.

Der Aufruf der Methode *visual_effect* erhält als ersten Parameter den Effekt, hier *:highlight*. Der Name des Effekts entspricht dabei der JavaScript-Methode (z.B. *:highlight* für *Effect.Highlight* oder *:drop_out* für *Effect.DropOut*).

Im zweiten Parameter wird die ID des hervorzuhebenden DOM-Objekts festgelegt. Wir greifen dabei auf die Instanzvariable *@task* zu, die in der Action *add_ajax_task* erzeugt wurde. Wie auch bei RHTML-Templates stehen uns alle Instanzvariablen aus der Action im RJS Template zur Verfügung. Die ID des DOM-Objekts zum Task ergibt sich aus dem Präfix *task_* und der Task-ID (siehe auch Listing 10.5).

Im dritten Parameter werden entsprechende Optionen übergeben, die den Effekt beeinflussen. Im Beispiel definieren wir die Dauer des Effekts auf drei Sekunden, damit wir den Zauber auch wirklich sehen.

Über die Methode *form* auf dem *page*-Objekt erhalten wir Zugriff auf die über die DOM-ID *add_task_form* referenzierte Form. Mit Hilfe der Methode *reset* setzen wir diese zurück. Dies geschieht über den RJS Class Proxy, den wir in Abschnitt 10.8.2 kennenlernen.

Die letzte Zeile im RJS Template dient der internen Aktualisierung der sortierten Liste aller Tasks. Würden wir den Code weglassen, so wäre der hinzugefügte Task nicht sofort per Drag and Drop sortierbar.

Vom Prinzip her war das schon alles. Wir haben mit Hilfe eines RJS Templates mehrere DOM-Objekte auf der Seite gleichzeitig aktualisiert. Dafür war kein JavaScript notwendig, da dieses von Rails erzeugt wird. Insgesamt gar nicht so schwer, oder?

Noch einmal zusammengefasst:

- Ajax-Methode ohne die Option *:update* aufrufen,
- die Action ein RJS Template rendern lassen und
- das RJS Template zur Action erstellen.

10.8.2 Weitere Techniken

Im folgenden Abschnitt wollen wir einige weitere Möglichkeiten von RJS kurz betrachten.

RJS im Controller

Sie können den Quellcode aus dem RJS Template auch direkt im Controller definieren. Dazu wird der Methode *render* die Option *:update* und der RJS-Block übergeben. In unserem Beispiel würde die Action dann wie folgt aussehen:

Listing 10.21: `app/controllers/tasks_controller.rb`

```
def add_ajax_task
  iteration = Iteration.find(params[:iteration_id])
  @task = Task.new(params[:task])
  iteration.tasks << @task
  render :update do |page|
    page.insert_html :bottom, 'list', :partial => 'task'
    page.visual_effect :highlight, "task_#{@task.id}",
                      :duration => 3.0
    page.form.reset 'add_task_form'
    page.sortable 'list', :url => { :action => "order_tasks" }
  end
end
```

Dies mag für kurzen Code oder zum Testen sinnvoll sein. Wir empfehlen Ihnen aber, den Code prinzipiell im Template zu lassen. Die Trennung zwischen Logik und View bleibt dadurch besser erhalten. Dies ist hilfreich, wenn die Arbeit für Logik und View von zwei verschiedenen Entwicklern getätigt wird und die Action neben RJS auch HTML oder XML liefern soll (vgl. Abschnitt 5.5).

Redirect per RJS

Statt JavaScript zurückzuliefern, kann im Template oder direkt im Controller auch ein Redirect auf eine andere Action oder eine andere Seite definiert werden:

```
def meine_redirect_action
  render :update do |page|
    # page.redirect_to "http://www.rapidwebdevelopment.de"
    page.redirect_to :controller => 'projects'
  end
end
```

Helper-Methoden und RJS

Auch Helper-Methoden können RJS-Code enthalten und haben Zugriff auf das *page*-Objekt, sofern sie innerhalb des *update*-Blocks aufgerufen werden. Im Beispiel aktualisieren wir die Zeit per RJS und nutzen die Helper-Methode im *update*-Block der Action:

```
# app/helpers/application_helper.rb
```

```

module ApplicationHelper
  def update_time
    page.replace_html 'time', Time.now.strftime("%H:%M:%S")
  end
end

# app/controllers/application.rb
def update_time
  render :update do |page|
    page.update_time
  end
end

# irgendein-view.html.erb
...
<div id="time">
  <%= Time.now.strftime("%H:%M:%S") %>
</div>
<%= periodically_call_remote(:frequency => 1,
  :url => { :action => "update_time" }) %>
...

```

RJS Class Proxy

Über den *RJS Class Proxy* haben wir per Ruby Zugriff auf die Klassenmethoden der JavaScript-Bibliotheken. Auf diese Weise können wir z.B. mit der Methode *form* auf die Klasse *Form* aus *prototype.js* und ihre Methoden zugreifen. In Listing 10.20 nutzten wir dies, um die Form zurückzusetzen.

Sofern die eigenen JavaScript-Dateien im Verzeichnis *public/javascripts* liegen (z.B. in *application.js*), können wir über den Class Proxy auch auf die JavaScript-Klassen in diesen Dateien zugreifen. Als Beispiel definieren wir uns eine Hinweis-Box:

```

var InfoBox = {
  message: function(text) {
    alert(text);
  }
}

```

Nun greifen wir über die durch den Class Proxy bereitgestellte Methode *info_box* auf die Methoden der JavaScript-Klasse *InfoBox* zu:

```

page.info_box.message('Hallo Rails')

```

Durch den RJS Class Proxy erreichen wir so eine Trennung zwischen dem JavaScript-Code in den Dateien und dem einfachen Zugriff darauf im Template. JavaScript kann somit auch in anderen Anwendungen (auch außerhalb von Rails) genutzt werden. Anders herum können Sie die Implementierung des JavaScripts austauschen, ohne den RJS-Code ändern zu müssen.

RJS Element Proxy

Um direkt auf ein Element im DOM-Baum zugreifen zu können, steht der *RJS Element Proxy* bereit. Dieser definiert die Methode `[[3`, der wir die ID des DOM-Elements übergeben. Im Beispiel erhalten wir Zugriff auf das Element *task_list* (vgl. Listing 10.4) und können entsprechende JavaScript-Methoden darauf aufrufen:

```
page['task_list'].hide
page[:task_list].hide.show
```

RJS Collection Proxy

Über den *RJS Collection Proxy* können Sie mehrere Elemente aus dem DOM-Baum ermitteln und manipulieren. Im Beispiel haben wir folgenden HTML-Code erstellt:

```
# app/views/rjs_proxy/index.html.erb
<div id="header">Kopf</div>
<div id="content">Inhalt
  <p>Erster Absatz</p>
  <hr>
  <p>Zweiter Absatz</p>
  <hr>
  <p>Dritter Absatz</p>
</div>
<div id="footer">Fußzeile</div>

<%= link_to_remote "Verstecke",
  :url => { :action => "collection_proxy", :hide => true } %>
<%= link_to_remote "Zeige",
  :url => { :action => "collection_proxy" } %>

# app/controllers/rjs_proxy_controller.rb
def collection_proxy
  @hide = params[:hide]
end
```

Der folgende Code sammelt mit Hilfe der Methode *select* alle Absätze auf, die innerhalb des *div*-Containers mit der ID *content* stehen, und liefert diese in einem Array zurück. Über Letzteres wird iteriert und jedes einzelne DOM-Element versteckt oder angezeigt:

```
# app/views/rjs_proxy/rjs_proxy_controller.rjs
page.select('#content p').each do |element|
  @hide ? element.hide : element.show
end
```

Der Aufruf `http://localhost:3000/rjs.proxy` zeigt das Verhalten.

³ Genauer gesagt, wird *JavaScriptGenerator* um die Methode erweitert.

10.8.3 Debugging und Testen

Wie wir im letzten Abschnitt gesehen haben, kommt man bei der Entwicklung von RJS Templates prinzipiell ohne explizite JavaScript-Programmierung aus. Das stimmt aber nur, solange es um die Programmierung selbst geht. Sobald nämlich der Code nicht macht, was Sie erwarten, wird es komplizierter.

Hier kommen Sie dann früher oder später nicht um das notwendige Wissen in Bezug auf JavaScript und den DOM-Baum herum. Sie können aber dennoch ohne große Kenntnisse starten und sich bei Bedarf nach und nach mit JavaScript vertraut machen. Das ist ohnehin nicht verkehrt, wenn Sie sich mit der Entwicklung von Webanwendungen eingehender beschäftigen müssen oder wollen.

Es gibt aber bereits erste Unterstützung für das Debugging und Testen von RJS, die wir kurz vorstellen wollen.

Fehler anzeigen

In der Entwicklungsumgebung (vgl. 11.1) wird der von Rails erzeugte JavaScript-Code in einen *try/catch*-Block eingeschlossen. Kommt es bei der Ausführung des JavaScripts im Browser zu einem Fehler, wird dieser über zwei *alert*-Boxen angezeigt. Die erste liefert die Fehlermeldung und die zweite den JavaScript-Code, der den Fehler verursachte. Sie können dieses Verhalten abstellen, indem Sie die Variable `config.action_view.debug_rjs` in der Datei `config/environments/development.rb` auf *false* setzen.

Debugging mit FireBug

Eine weitere sehr gute Hilfe, Fehlern auf die Schliche zu kommen, ist das unter www.getfirebug.com zu findende Add-on *Firebug* für den Firefox Browser⁴. Es ermöglicht Ihnen die Analyse des DOM-Baums bis ins Detail. Sie können z.B. Ausgaben in Ihren Webseiten direkt in die Firebug-Konsole loggen oder Ihr JavaScript Schritt für Schritt debuggen. Im Falle von Ajax werden die Requests und Responses geloggt, sodass Sie den tatsächlich zurückgelieferten JavaScript Code analysieren können und eventuelle Fehler leichter finden.

Auch wenn Sie grundsätzlich nicht mit Firefox arbeiten, lohnt sich die Nutzung des Browsers mit dem Add-on während der Entwicklung. Eine Beschreibung würde den Rahmen des Buches sprengen. Informationen finden sich auf der Add-on-Homepage oder im PDF-Dokument [22]. In naher Zukunft wird es für andere Browser mit Sicherheit ähnliche Plugins geben.

Testen mit ARTS

Wie für andere Bereiche von Rails ist es auch bei der Verwendung von RJS wünschenswert, die korrekte Funktionsweise der RJS Templates durch Tests prüfen zu können. (Eine Einführung in das Testen Ihrer Anwendung mit Rails finden Sie in

⁴ www.mozilla-europe.org/de/products/firefox

Kapitel 14.) Hier kommt uns das Rails Plugin *ARTS* zu Hilfe, das wir kurz vorstellen wollen.

Zur Installation folgen Sie den Schritten nach der Anleitung unter der Webseite <http://glu.ttono.us/articles/2006/05/29/guide-test-driven-rjs-with-arts> und installieren das Plugin:

```
$ script/plugin install arts
```

In der Klasse *test/functional/tasks_controller.test.rb* erstellen wir exemplarisch folgenden Test für unsere Methode *add_ajax_task*:

```
...
class TasksControllerTest < ActionController::TestCase
  ...
  def test_add_ajax_task
    xhr :post, :add_ajax_task,
      { :iteration_id => iterations(:one).id,
        :task => { :name => "neuer_task" } },
      { :person => people(:one) }
    assert_rjs :insert_html, :bottom, 'list'
    assert_rjs :visual_effect, :highlight,
      "task_#{assigns(:task).id}", :duration => 3.0
  end
end
```

Der Test sendet mit Hilfe der Methode *xhr* einen Post-Request an die zu testende Methode, wobei wir die notwendigen Parameter und einen Anwender in der Session übertragen.

Die Methode *assert_rjs* ermöglicht nun, den von Rails gelieferten JavaScript-Code zu testen. Wir prüfen, ob der *insert_html*-Aufruf den HTML-Code für den neuen Task am Ende (*:bottom*) der Liste (ID *list*) einfügt. Ferner, dass der neu hinzugefügte Task per *:highlight* drei Sekunden lang hervorgehoben wird.

Über *assert_rjs* sind weitere Tests möglich. Informieren Sie sich hierzu auf der ARTS-Homepage.

10.9 Zusammenfassung und Ausblick

Dieses Kapitel hat einen ersten Einblick in die Entwicklung von Web-Anwendungen mit Ajax geliefert. Letztendlich sind Ajax und die anderen in diesem Kapitel beschriebenen Funktionen und Effekte „nur“ eine Reihe weiterer Werkzeuge für die Entwicklung „reicher“ Web-Anwendungen.

Die nächste und eigentliche Herausforderung ist die Suche nach neuen Ideen für kreative, benutzerfreundliche und noch nie dagewesene Web-Anwendungen, um diese mit den jetzt zur Verfügung stehenden Technologien Wirklichkeit werden zu lassen. Der Einsatz von Ajax sollte dabei aber wohlüberlegt sein und behutsam geschehen, denn er bietet nicht nur Vorteile. Einen interessanten Artikel über die Gefahren von Ajax finden Sie unter <http://www.devx.com/webdev/Article/28861>.

Kapitel 11

Produktion

In diesem Kapitel geben wir Ihnen einige Tipps und Hinweise zum Betrieb einer Rails-Anwendung in Produktion. Dabei konzentrieren wir uns auf das Betriebssystem Linux und die Kombination aus Apache und Mongrel, da es sich um das wohl gängigste Setup handelt. Wie Sie Ihre Anwendung in Produktion mit Hilfe von Capistrano ausliefern, beschreiben wir in Kapitel 12.

11.1 Umgebungen in Rails

Rails unterscheidet die drei Umgebungen *development* (Entwicklung), *test* (Test) und *production* (Produktion). Dadurch wird es zum einen möglich, pro Umgebung eine Datenbank bereitzustellen, sodass Änderungen an den Daten in einer Umgebung nicht die Daten einer anderen beeinflussen. Für die Testfälle sind z.B. entsprechende Datenkonstellationen notwendig, die über die Fixtures in die Datenbank geschrieben werden (vgl. Abschnitt 14.2). Diese Daten dürfen die Entwicklungsdaten nicht überschreiben. Ebenso dürfen Änderungen an den Entwicklungsdaten nicht zu Fehlern bei den Unit Tests führen. Zum anderen werden je nach Umgebung spezifische Einstellungen aktiviert.

11.1.1 Umgebung definieren

Rails unterscheidet die drei Umgebungen am Wert der Umgebungsvariable `RAILS_ENV`. Ist die Umgebungsvariable nicht definiert, verwendet Rails initial *development*. In Abhängigkeit von `RAILS_ENV` lädt Rails die Einstellungen aus der umgebungsspezifischen Datei `config/environments/<RAILS_ENV>.rb`. Im Beispiel von `config/environments/development.rb` enthält die Datei z.B. folgende Einträge:

Listing 11.1: `config/environment/development.rb`

```
# Settings specified here will take precedence over those in
# config/environment.rb
```

```
# In the development environment your application's code is
# reloaded on every request. This slows down response time but is
# perfect for development since you don't have to restart the web
# server when you make code changes.
config.cache_classes = false

# Log error messages when you accidentally call methods on nil.
config.whiny_nils = true

# Show full error reports and disable caching
config.action_controller.consider_all_requests_local = true
config.action_view.debug_rjs                         = true
config.action_controller.perform_caching             = false
config.action_view.cache_template_extensions         = false

# Don't care if the mailer can't send
config.action_mailer.raise_delivery_errors = false
```

Beachten Sie, dass Einstellungen in den umgebungsspezifischen Dateien wie *config/environments/development.rb* die Einstellungen in der allgemeinen Konfigurationsdatei *config/environment.rb* überschreiben.

11.1.2 Entwicklung

Während der Entwicklung spielt die Performanz der Anwendung keine Rolle. Statt dessen stehen einfache Nutzung und schnelles Feedback bei Änderungen im Vordergrund. Daher wird in der Entwicklung über *config.cache_classes = false* definiert, dass die Anwendung für jeden Request neu geladen wird. Dadurch werden Änderungen am Quellcode durch ein einfaches Reload im Browser sofort sichtbar. Für ein schnelles Feedback bei Änderungen ist zudem das Caching über *perform_caching = false* ausgeschaltet.

Fehler müssen sofort sichtbar sein und möglichst viele Kontextinformationen enthalten. Die Einstellung *consider_all_requests_local = true* sorgt dafür, dass alle Requests als lokale Requests interpretiert werden. Dies hat zur Folge, dass im Fehlerfall ein View mit Stacktrace und Kontextinformationen ausgeliefert wird.

11.1.3 Test

Für einen Testlauf ist wiederum die Performanz, d.h. die schnelle Durchführung der Tests, wichtig. Die Anwendung ist einmalig zu laden (*config.cache_classes = true*), da es während des Testlaufs keine Änderungen am Quellcode gibt. Die anderen Einträge sind analog zur Entwicklungsumgebung definiert.

11.1.4 Produktion

Für die Produktion ist hingegen neben Punkten wie Stabilität, Sicherheit usw. die Performanz von entscheidender Bedeutung. Daher wird das erneute Laden der An-

wendung über `config.cache_classes = true` unterbunden. Ebenso ist das Caching von Views über `perform_caching = true` aktiviert.

Alle Requests werden in Produktion als entfernt interpretiert (`consider_all_requests_local = false`). Dadurch wird dem Anwender im Fehlerfall eine Fehlerseite angezeigt. Eine Ausnahme bilden alle Requests, die von einer der IP-Adressen stammen, die in der Methode `ActionController::Base.local_request?` definiert sind. Per Default ist dies nur die IP des lokalen Rechners (`127.0.0.1`). Durch das Überschreiben der Methode im `ApplicationController` können weitere Entwicklungsrechner hinzugefügt werden:

```
def local_request?  
  ["127.0.0.1",  
   "<ip-meines-entwicklungsrechners>",  
   ... ].include?(request.remote_ip)  
end
```

Per Default zeigt Rails in Produktion bei einem falschen Routing oder einer unbekannten Action die Seite `public/404.html` an. Kann ein Datensatz nicht aktualisiert werden, weil er nicht valide (`ActiveRecord::RecordInvalid`) oder das Authenticity Token (vgl. Abschnitt 6.3.2) ungültig ist, so wird die Seite `public/422.html` angezeigt. Alle anderen Fehler führen zur Anzeige der Seite `public/500.html`. Vergessen Sie nicht, diese Seiten für die Produktion bezüglich Layout, Sprache usw. anzupassen.

Um über interne Fehler per E-Mail informiert zu werden, nutzen Sie am besten das Plugin `ExceptionNotifier` (vgl. Abschnitt 11.9).

11.1.5 Umgebung hinzufügen

Wir möchten die neuen Features der Anwendung unseren Kunden präsentieren, bevor die Version in Produktion scharfgeschaltet wird. Die Anwendung soll sich dabei wie in Produktion verhalten, damit unter realen Bedingungen getestet und geprüft werden kann. Dazu benötigen wir eine Art Akzeptanztest-Umgebung, die häufig als *Staging*¹ bezeichnet wird.

Dazu erstellen wir in der `config/database.yml` einen Datenbankeintrag für *staging*:

Listing 11.2: `config/database.yml`

```
...  
staging:  
  adapter: mysql  
  database: ontrack_staging  
  username: ontrack  
  password: geheim  
  host: localhost
```

Danach erzeugen wir eine neue Datei `config/environments/staging.rb`, kopieren die Einstellungen aus `config/environments/production.rb` hinein und passen sie ggf. an.

¹ zu Deutsch: Bühne, im Sinne einer Präsentation

Zum Schluss legen wir die Datenbank *ontrack_staging* an und erzeugen das Schema per Migration. Hierbei geben wir per *RAILS_ENV* die Umgebung *staging* an:

Listing 11.3: config/database.yml

```
$ rake db:create RAILS_ENV=staging
$ rake db:migrate RAILS_ENV=staging
```

Ebenso ist die Umgebung in der Konfiguration des Webservers unter *environment* einzutragen (vgl. Listing 11.5). Nun können wir die Datenbank mit realistischen Testdaten füllen und nutzen. Wir verwenden hier aber keine Daten aus Produktion, da sonst die Gefahr besteht, dass wir z.B. E-Mails an Adressen verschicken, die nur in Produktion gültig sind.

11.2 Webserver und Datenbank

Um Ihre Anwendung in Produktion zu bringen, benötigen Sie zunächst einen oder mehrere Rechner, auf denen die Anwendung laufen soll. Auf ihnen müssen Ruby, RubyGem, Rails, eventuell zusätzlich benötigte Gems, die Datenbank und der Webserver installiert sein. Eine Liste möglicher Provider finden Sie u.a. im Wiki der Rails Usergroup Deutschland <http://www.rubyonrails-ug.de/wiki/rails-hoster>. Je nach Provider können Sie sich einige der folgenden Installationsschritte sparen, da sie bereits vom Anbieter übernommen wurden.

11.2.1 Datenbank

Bezüglich der Datenbank gibt es für Rails keine besonderen Anforderungen, sodass wir hierauf nicht weiter eingehen. Für den Einsatz im kommerziellen Umfeld können ggf. Lizenzkosten entstehen, die Sie bei Ihrer Entscheidung berücksichtigen sollten. Möglicherweise stellt Ihnen der Provider bereits die Datenbank zur Verfügung.

11.2.2 Webserver Cluster

Das Rails-Framework ist nicht für die Verwendung von Threads konzipiert. Es kann zu einem Zeitpunkt nur von einem Prozess durchlaufen werden. Für die Entwicklung und den Test ist das kein Problem. Für die Produktion mit vielen parallelen Anfragen würde ein einziger Prozess zur Zeit alle anderen Anfragen warten lassen und die Anwendung langsam und nicht nutzbar machen.

Daher wird eine Rails-Anwendung in Produktion von mehreren Instanzen eines Webservers gleichzeitig ausgeliefert. Die Anwendung wird dabei in einem einzigen Verzeichnis bereitgestellt, das von allen Instanzen gelesen wird. Jede Instanz wird zu einem Zeitpunkt von genau einem Prozess durchlaufen, der genau eine Datenbankverbindung hält.

Jeder Prozess nimmt die HTTP-Requests an einem anderen Port entgegen. Die Gruppe der Webserver-Instanzen wird als Cluster bezeichnet. Eine der Gruppe vorge-

schaltete Instanz (Proxy) nimmt die Anfragen am HTTP-Standard-Port 80 entgegen und leitet die Anfragen an die einzelne Instanzen weiter. Der Algorithmus für die Verteilung kann von einfach bis intelligent alles abdecken. Neben der entsprechenden Performanz und Lastverteilung wird auf diese Weise auch eine Ausfallsicherheit gewährleistet, falls einmal eine Instanz nicht zur Verfügung steht.

Aus diesem Grund finden Sie für die Produktion immer Kombinationen aus dem Proxy (z.B. Apache, Nginx, Pound) und den hintergelagerten Webserver-Instanzen (z.B. Mongrel).

Sie finden keine Kombinationen mit WEBrick, da dieser sehr langsam und speicherhungrig ist und sich nicht für den Einsatz in Produktion eignet. Versuchen Sie es erst gar nicht. Im Folgenden beschreiben wir eine verbreitete Kombination.

11.2.3 Apache und Mongrel

Zum Betreiben von produktiven Web-Anwendungen in Ruby haben sich Apache² 2.2 und Mongrel³ etabliert.

Apache ist dabei für die Auslieferung des statischen Inhalts Ihrer Anwendung, d.h. Bilder, Stylesheets, JavaScript-Dateien, PDFs usw. zuständig. Allen anderen Anfragen leitet er an Mongrel (bzw. den Cluster) weiter, der die Anwendung enthält und für die Auslieferung der dynamischen Seiten sorgt.

Die Installation einer Rails-Anwendung unter Apache 2.2 und Mongrel kann von System zu System unterschiedlich sein. Alle erdenklichen Konstellationen bzw. die grundsätzliche Konfiguration von Apache oder Mongrel zu beschreiben, liegt außerhalb dieses Buches. Wir setzen Grundkenntnisse in der Konfiguration von Apache voraus und orientieren uns an einer Installation unter Debian.

Als Erstes ist der Mongrel-Server wie folgt zu installieren:

```
$ sudo gem install daemons gem_plugin  
mongrel mongrel_cluster --include-dependencies
```

Für die Verwendung unter Apache müssen die folgenden Module in Apache eingebunden sein:

```
proxy.conf  
proxy.load  
proxy_connect.load  
proxy_http.load  
proxy_balancer.load
```

Dazu sind die Module ggf. unter dem Verzeichnis */etc/apache2/mods-available* zu installieren und aus */etc/apache2/mods-enabled* mit einem Symlink zu versehen, z.B.:

```
cd /etc/apache2  
$ ll mods-available/proxy_balancer.load  
... mods-available/proxy_balancer.load
```

² <http://apache.org>

³ <http://mongrel.rubyforge.org/>


```
$ ll mods-enabled/proxy_balancer.load
... mods-enabled/proxy_balancer.load ->
    ../mods-available/proxy_balancer.load
```

Wir erstellen einen virtuellen Host in der Datei `/etc/apache2/site-available/ontrack.conf`, der wie folgt aussieht:

Listing 11.4: `/etc/apache2/site-available/ontrack.conf`

```
<VirtualHost *:80>
    ServerName ontrack.b-simple.de
    DocumentRoot /var/www/ontrack/current/public
    ErrorLog      /var/log/apache2/ontrack_error.log
    CustomLog     /var/log/apache2/ontrack_access.log combined

    <Directory /var/www/ontrack/current/public>
        Options ExecCGI FollowSymLinks
        AllowOverride all
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>

<Proxy balancer://ontrack_cluster>
    BalancerMember http://127.0.0.1:8000
    BalancerMember http://127.0.0.1:8001
    BalancerMember http://127.0.0.1:8002
</Proxy>
```

Der Eintrag *DocumentRoot* beschreibt das Basisverzeichnis der Anwendung, aus dem Apache den statischen Inhalt ausliefert. Hierbei handelt es sich um das *public*-Verzeichnis des Rails-Projekts. Hier liegen alle statischen Seiten, wie z.B. *404.html* und in Unterverzeichnissen die Stylesheets-, JavaScript- und Image-Dateien.

Neben dem *DocumentRoot* definieren wir einen Mongrel-Cluster mit drei Instanzen, die an Port 8000, 8001 und 8002 laufen. Der Port und die Anzahl sind auch in der Mongrel-Konfigurationsdatei anzugeben (s. Listing 11.5). Die optimale Anzahl der Instanzen hängt von der jeweiligen Anwendung ab. Starten Sie mit 2-4-Instanzen, und analysieren Sie den Bedarf im Laufe des Projekts (vgl. Abschnitt 11.5).

Der Host wird durch den Symlink `/etc/apache2/site-enabled/ontrack.conf` auf die Datei aktiviert:

```
$ cd site-enabled
$ ln -s ../site-available/ontrack.conf ontrack.conf
$ cd ..
$ ll site-enabled/ontrack.conf -> ../site-available/ontrack.conf
```

Die Konfigurationsdatei für Mongrel bzw. den Cluster wird im Rails-Projektverzeichnis (nicht unter Apache!) unter `config/mongrel_cluster.yml` abgelegt und sieht im Beispiel wie folgt aus:

Listing 11.5: config/mongrel_cluster.yml

```
---
user: ontrack
group: ontrack
cwd: /var/www/ontrack/current
port: "8000"
environment: production
address: 0.0.0.0
pid_file: /var/run/mongrel_cluster/mongrel.pid
servers: 3
```

Für den Benutzer und die Gruppe wurde *ontrack* definiert. Ein entsprechender Benutzer inklusive der Gruppe ist auf dem Server anzulegen. Unter diesem laufen die Mongrel-Prozesse, die in der Regel über Capistrano (vgl. Kapitel 12) gestartet werden.

Wenn Sie die Prozesse einmal direkt auf dem Server per Hand starten, achten Sie darauf, dass Sie dies unter dem Benutzer tun, mit dem Sie auch über Capistrano arbeiten. Wenn Sie die Prozesse z.B. unter dem Benutzer *root* starten, haben Sie später bei einem Deployment mit Capistrano nicht das Recht, die Mongrel-Prozesse neu zu starten, und die Aktualisierung schlägt fehl.

Der Apache-Prozess ist bei einer Aktualisierung der Anwendung nicht neu zu starten. Er läuft daher nicht unter dem Anwender *ontrack*, sondern unter *root*, *www-data* oder einem anderen Benutzer.

Die Umgebung der Rails-Anwendung wird per *environment* auf *production* gesetzt, wodurch die produktionsrelevanten Konfigurationseinträge Verwendung finden (vgl. Abschnitt 11.1.4).

Unter *cwd* wird derselbe Pfad wie unter *DocumentRoot* in der *apache2.conf*-Datei angegeben, und es werden durch *servers* genau die drei Instanzen aus *apache2.conf* ab Port 8000 (*port*) definiert.

Die Prozessdatei jeder Instanz wird unter */var/run/mongrel_cluster* erzeugt. Sie müssen das Verzeichnis anlegen, falls es nicht existiert. Nach einem erfolgreichen Start der Instanzen wird die Portnummer in den Dateinamen aufgenommen, und ein Blick in das Verzeichnis liefert Folgendes:

```
$ ll /var/run/mongrel_cluster/
...mongrel.8000.pid
...mongrel.8001.pid
...mongrel.8003.pid
```

Falls Sie weitere Anwendungen haben, lassen Sie die Pid-Dateien ebenfalls hier erstellen. So haben Sie alle Mongrel-Prozesse im Blick. Gegebenenfalls benennen Sie die Dateien nach der Anwendung, z.B. *ontrack.pid* und *cms.pid*.

In der Datei *public/.htaccess* zu Ihrem Projekt sind die Rewrite Rules entsprechend zu setzen, so dass statischer Inhalt von Apache und alles andere von Mongrel ausgeliefert wird:

Listing 11.6: public/.htaccess

```

RewriteEngine On
...
# Redirect all non-static requests to cluster
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ balancer://ontrack_cluster%{REQUEST_URI} [P,QSA,L]

# Error pages
ErrorDocument 500 /500.html
ErrorDocument 404 /404.html
ErrorDocument 403 /403.html

```

Hier ist derselbe Name (*ontrack_cluster*) anzugeben, den Sie auch in der *apache2.conf* verwendet haben. Ebenso werden die Fehlerseiten definiert.

Für das Deployment mit Capistrano müssen Sie die Konfigurationsdatei *deploy.rb* anpassen, damit Capistrano die entsprechenden Kommandos für das Starten und Stoppen der Mongrel-Prozesse kennt:

Listing 11.7: config/deploy.rb

```

require 'mongrel_cluster/recipes'
...

```

Neben dem Start der Mongrel-Instanzen über Capistrano kann der Cluster auch über das Skript *mongrel_rails* und unter dem Benutzer *ontrack* erfolgen:

```

ontrack $ mongrel_rails cluster::start
        -C /var/www/ontrack/current/config/mongrel_cluster.yml

```

Damit ist die Konfiguration abgeschlossen. Starten Sie den Apache, und führen Sie ein Deployment der Anwendung mit Capistrano durch. Unter der URL des virtuellen Hosts (im Beispiel *ontrack.b-simple.de*) sollte jetzt Ihre Anwendung laufen.

Ist dies nicht der Fall, prüfen Sie die Log-Ausgaben von Apache (*ontrack_error.log* und *ontrack_access.log*), Mongrel (*mongrel.8000.log*) und der Anwendung (*production.log*). Kommt der Request jeweils an, und gibt es Fehler?

Neben der Mongrel-Homepage <http://mongrel.rubyforge.org> bietet auch die Seite <http://blog.codahale.com/2006/06/19/> weitere Informationen zur Installation und Konfiguration von Apache und Mongrel.

Die Installation ist nicht ohne, verzweifeln Sie also nicht gleich, wenn es nicht auf Anhieb läuft. Es gibt mittlerweile viele Informationen zu unterschiedlichen Konfigurationen im Internet. Wenn es gar nicht klappt, ist auch über die Mailingliste der Rails Usergroup <http://www.rubyonrails-ug.de> Hilfe zu bekommen. Viel Erfolg!

11.2.4 Alternativen

Als Alternative zum eher schwergewichtigen Apache erfreut sich *Nginx*⁴ zunehmender Beliebtheit. Wie Apache liefert auch er die statischen Inhalte aus und dient als Proxy. Nginx gilt als sehr schnell und einfach zu konfigurieren.

⁴ <http://nginx.net>

Weitere mögliche Kombinationen sind Apache und FastCGI⁵ oder Lighttpd⁶ und FastCGI oder auch LiteSpeed⁷. Darüber hinaus gibt es sicher weitere Webserver und Kombinationen, deren Betrachtung aber außerhalb dieses Buches liegt. Vielleicht haben Sie ja mal ein wenig Zeit zum Probieren.

11.3 Skalierung

In diesem Abschnitt gehen wir kurz auf die Skalierung einer Rails-Anwendung ein. Der Begriff *Skalierbarkeit* beschreibt die Eigenschaft eines Systems, auch bei zunehmender Systemlast den erwarteten Grad an Servicequalität zu erbringen, ohne dass die Architektur des Systems dafür angepasst werden muss.

Rails-Anwendungen werden horizontal skaliert, d.h. bei zunehmender Systemlast wird die Anzahl der für den Betrieb benötigten Server erhöht (siehe Abbildung 11.1).

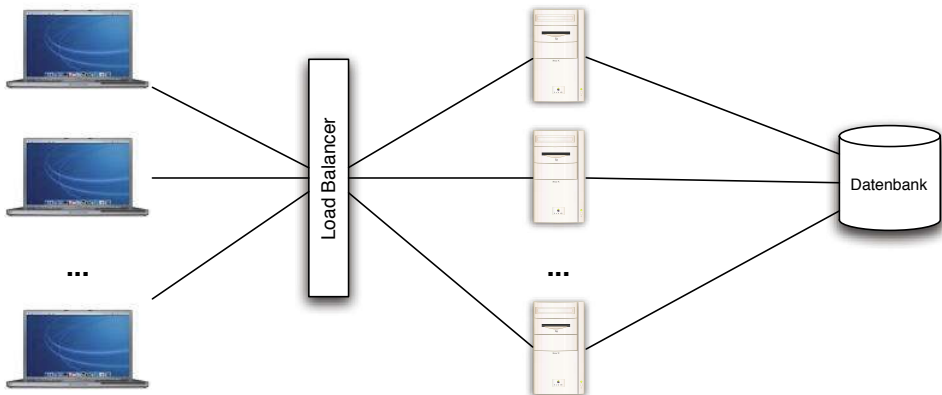


Abbildung 11.1: Bei steigender Systemlast wird die Anzahl der Server erhöht.

Horizontale Skalierbarkeit setzt voraus, dass die Anwendung keine Request-überdauernden Daten in der Web- oder Anwendungsschicht speichert. Die Requests einer Benutzersession bleiben dadurch zusammengehörig, wobei es egal ist, welcher Server die einzelnen Requests bearbeitet.

Die einzigen Schnittstellen zusammengehöriger Requests sind die HTTP-Session, die Datenbank und ggf. das Caching-Verzeichnis. Für Datenbanken gibt es von Rails auch keine besonderen Anforderungen. Die Konfiguration des Session-Speichers wird im nächsten Abschnitt 11.4 beschrieben.

⁵ <http://www.fastcgi.com/>

⁶ <http://www.lighttpd.net>

⁷ <http://www.litespeedtech.com/overview-2.html>

11.4 Session-Speicher

Per Default speichert Rails Session-Daten in einem Cookie. Aus Sicht der Skalierung ist diese Einstellung optimal. Egal, welcher Server aus dem Cluster die Anfrage bearbeitet, die notwendigen Session-Daten werden beim Request mitgeliefert, und der Server muss sich nicht um die Verwaltung kümmern. Der Cookie selbst kann zwar nur eine begrenzte Menge von Daten aufnehmen (4 KB), doch ist es in jedem Fall nicht empfehlenswert, große Daten in der Session zu halten (vgl. Abschnitt 5.8).

Die Session-Daten im Cookie sind verschlüsselt, können aber bei einem zu einfachen Passwort und mit entsprechendem Aufwand entschlüsselt werden. Wir empfehlen Ihnen daher, keine schützenswerten Daten in der Session zu speichern. Auch hier gilt, keine Objekte in der Session zu speichern. Knackt der Angreifer die Verschlüsselung, kann er ein modifiziertes Objekt an den Server schicken. Dadurch verschafft er sich Zugriff auf Teile der Anwendung, für die er nicht autorisiert ist.

Solange es sich nur um die ID des angemeldeten Benutzers oder um Flash-Meldungen etc. handelt, ist das kein Problem. Andere Informationen sollten Sie ohnehin nicht in der Session speichern.

Das Passwort für die Verschlüsselung sollte mehr als 40 Zeichen enthalten und wird serverseitig unter *secret* in der Datei *config/environment.rb* definiert:

Listing 11.8: *config/environment.rb*

```
Rails::Initializer.run do |config|
  ...
  config.action_controller.session = {
    :session_key => '_ontrack_session',
    :secret      => '3da56b7db7412da1.....'
  }
```

Möchten Sie statt der Cookies die Session-Daten in der Datenbank speichern, aktivieren Sie in der *config/environment.rb* folgenden Eintrag:

Listing 11.9: *config/environment.rb*

```
Rails::Initializer.run do |config|
  ...
  config.action_controller.session_store = :active_record_store
```

Um die notwendige Tabelle *sessions* in der Datenbank anzulegen, nutzen Sie den folgenden Rake-Task, der ein Migrationsskript (vgl. Abschnitt 3.5.2) erzeugt, das Sie im Anschluss ausführen:

```
$ rake db:sessions:create
...
$ rake db:migrate
```

In diesem Fall wird die Session-Id über ein Cookie bei jedem Request mitgeliefert, und der Server ermittelt darüber die Sitzungsdaten aus der Datenbank. Eine performanzoptimierte Version des *ActiveRecordStore* finden Sie bei Bedarf unter <http://railsexpress.de>.

Alternativ steht Ihnen mit *MemCached*⁸ die Speicherung im Hauptspeicher eines Servers zur Verfügung, was die Performanz erhöhen kann. Speichern Sie aus Gründen der Performanz niemals die Session-Daten im Dateisystem.

Beachten Sie, dass Rails ausschließlich mit Cookies arbeitet und Browser und clientseitige Firewall daher das Speichern von Cookies auf Ihrem Rechner erlauben müssen.

11.5 Tipps zur Performanz

Der Begriff *Performanz* beschreibt das Antwortzeitverhalten eines Systems, bezogen auf die Abarbeitung von Requests und Darstellung der zugehörigen HTML-Views.

Die Performanz ist sicher ein wichtiger Aspekt einer Anwendung, und Sie sollten das Thema nicht einfach mit mehr und schnellerer Hardware erschlagen. Aber solange Sie nicht das nächste Amazon oder Ebay schreiben, sollte Ihnen die Performanz Ihrer Anwendung keine schlaflosen Nächte bereiten.

Berücksichtigen Sie bei der Programmierung Ihrer Anwendung einige grundlegende Punkte, und nehmen Sie sich dem Thema erst mit steigendem Bedarf an. Unter der Webseite <http://railsexpress.de> finden Sie eine Reihe von Tipps für typischen Performanzkiller in Rails-Anwendungen. Ebenso entsprechende Werkzeuge für die Performanzanalyse.

Kommt es zu Performanzengpässen, analysieren Sie die Flaschenhälse Ihrer Anwendung und versuchen Sie nicht, ziellos zu optimieren. In der Regel stecken 80% der Verarbeitungszeit in 20% der Anwendung. Finden Sie die performanzkritischen Stellen, und optimieren Sie hier.

Prüfen Sie die SQL-Statements im Log. Können Sie die Anzahl der Datenbankaufrufe verringern (vgl. Abschnitt 4.5.6). Haben Sie langlaufende Prozesse, die nicht über den Webserver, sondern im Hintergrund ausgeführt werden sollten? Vielleicht per Cron-Job oder mithilfe von *BackgroundRb*⁹. Ebenso kann das Caching der Seiten einen Performanzgewinn bringen, auf das wir in Abschnitt 6.9 eingehen.

Die richtige Anzahl der Mongrel-Instanzen hat sicher einen Einfluss auf die Performanz. Zu wenige Instanzen können diese senken, zu viele jedoch auch. Die optimale Anzahl in Kombination mit anderen Performanz-Optimierungen zu finden, ist eine Menge Arbeit. Die Online-Dokumentation zu Mongrel liefert einige Tipps.

In der unter 11.2.3 beschriebenen Konfiguration ist die Anzahl der Mongrel-Instanzen fest definiert und lässt sich ohne Neustart der Prozesse auch nicht ändern. Kommt es zu einer temporären Erhöhung der Last auf dem System, wären weitere Mongrel-Instanzen hilfreich. Sie können gerade jetzt aber schlecht die Server stoppen, um die Konfiguration zu ändern. Besser wäre, die Instanzen dynamisch zu erhöhen und zu verringern. Hier kann Ihnen *Swiftly*¹⁰ helfen.

⁸ <http://www.danga.com/memcached>

⁹ <http://backgroundrb.rubyforge.org/>

¹⁰ <http://swiftply.swiftcore.org>

Wenn Ihre Anwendung dann immer populärer wird¹¹ und die Last auf den Servern temporäre Spitzen erreicht, die das Setup so nicht bearbeiten kann, können Sie auch über den Einsatz von Amazon S3 und EC2 nachdenken (<http://aws.amazon.com>).

11.6 Sicherheit

Mit Sicherheit ist hier der Schutz vor unbefugtem Zugriff auf die Anwendung und Ihrer Daten gemeint. Im Folgenden geben wir einige kurze Tipps, um die Sicherheit einer Rails-Anwendung zu erhöhen. Die Liste erhebt keinen Anspruch auf Vollständigkeit. Weitere wichtige Informationen zur Sicherheit von Rails-Anwendungen finden Sie u.a. auch unter <http://www.rorsecurity.info>.

Zugriff auf Datensätze

Der Zugriff auf einen Datensatz erfolgt in der Regel über die ID, die in der URL angegeben wird, z.B. <http://meinedomain.de/projects/42>. Sofern Sie die Daten einfach über `Project.find(params[:id])` ermitteln, hat ein Benutzer jederzeit Zugriff auf die Projektdaten eines anderen Benutzers. Er muss dazu nur eine andere ID in der URL angeben. Stellen Sie sicher, dass ein Benutzer nur die Datensätze sieht, für die er auch autorisiert ist. Wie Sie dies tun können, haben wir in Abschnitt 5.6 beschrieben.

Mass Assignment

Die Formular-Helper mit Bezug zu Modellen sind eine feine Sache, da sie die Zuweisung von Daten aus einem Formular an ein Modell in einem Schritt ermöglichen. Denken Sie daran, schützenswerte Attribute (z.B. das Admin-Flag) von der Massenzuweisung auszuschließen. Wir haben dies in Abschnitt 11.6 beschrieben.

SQL-Injection

Verhindern Sie die Nutzung von unmaskierten Daten beim Zugriff auf die Datenbank (SQL-Injection¹²), indem Sie durchgängig die Array-Notation verwenden (vgl. Abschnitt 4.5.2).

Cross-Site Scripting (XSS)

Beim Cross Site Scripting, kurz XSS, wird HTML oder JavaScript von böswilligen Benutzer in die Seiten anderer Benutzer eingeschleust.¹³ Sie umgehen dies, indem Sie die Methoden `html_escape` (vgl. Abschnitt 6.3.14) und `sanitize` (vgl. Abschnitt 6.3.14) beim Einfügen von dynamischen Daten in den View nutzen. Schauen Sie noch einmal in den entsprechenden Abschnitten nach, und prüfen Sie Ihre Views.

¹¹ Was wir Ihnen natürlich wünschen.

¹² <http://de.wikipedia.org/wiki/SQL-Injection>

¹³ http://de.wikipedia.org/wiki/Cross-Site_Scripting

Cross Site Request Forgery (CSRF)

Wie wir in 6.3.2 beschrieben haben, unterstützt Rails Sie beim Verhindern von Cross Site Request Forgery. Prüfen Sie nochmals Ihre Formulare.

GET vs. POST

Stellen Sie sicher, dass alle GET-Requests an Ihre Anwendung idempotent sind, d.h. durch sie keine Änderungen am Zustand des Systems erfolgt. Wir haben diesen Punkt im Abschnitt 5.4 beschrieben.

Stellen Sie ebenso sicher, dass nach einem POST-Request an Ihre Anwendung immer ein Redirect auf eine neue URL durchgeführt wird. Wir haben diese Regel im Abschnitt 5.3 aufgeführt. Einen interessanten Artikel zum Thema finden Sie unter <http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPost>.

Subversion Meta-Daten

Führen Sie beim Deployment mit Capistrano ein *Checkout* der Sourcen aus dem Subversion-Repository aus, werden die Repository-Informationen im Unterverzeichnis `.svn` eines jeden Verzeichnisses abgelegt. Der Zugriff über die URL <http://meinedomain.de/.svn/entries> liefert dann diese Daten.

Sie können die Auslieferung verhindern, indem Sie das Deployment über die Subversion-Methode `Export` (`export`) durchführen. Im Falle von Capistrano definieren Sie den Eintrag wie folgt:

Listing 11.10: config/deploy.rb

```
..
set :repository, "http://..."
set :checkout, "export"
...
```

Dadurch wird kein Verzeichnis `.svn` erzeugt. Alternativ schützen Sie im Apache den Zugriff auf die `.svn`-Verzeichnisse wie folgt:

Listing 11.11: /etc/apache2/site-available/ontrack.conf

```
<VirtualHost *:80>
...
<DirectoryMatch "^/.* /\.svn/">
    ErrorDocument 403 /404.html
    Order allow,deny
    Deny from all
    Satisfy All
</DirectoryMatch>
</VirtualHost>
```

Passwörter

Für eine Rails-Anwendung werden Passwörter für die Datenbank, den Session-Speicher, den SMTP-Server usw. per Default im Klartext definiert. Die Projektdatei-

en werden in einem Versionskontrollsystem verwaltet, auf das ggf. viele Entwickler und andere Personen Zugriff haben. Auf diese Weise werden Passwörter für Produktion verbreitet und leicht zugänglich für Unbefugte gemacht.

Um dieses Problem zu umgehen, können die Passwörter für Produktion nur auf dem Produktionsserver in entsprechenden Dateien (z.B. *secret/database.yml*) abgelegt werden, die beim Deployment die Originaldatei aus dem Repository ersetzen.

Alternativ wird in der Konfigurationsdatei das Passwort aus einer Passwort-Datei geladen, die nur auf dem Produktionsserver vorhanden ist. Im Beispiel wird beim Einlesen der Konfigurationsdatei *config/database.yml* das Datenbank-Passwort über den Aufruf von *PasswordSafe.get* aus der Datei */home/ontrack/.pwd* gelesen:

Listing 11.12: *config/database.yml*

```
production:
  adapter: mysql
  database: ontrack_production
  host: localhost
  username: ontrack
  password: <%= PasswordSafe.get(:ontrack_production_db) %>
...

# /home/ontrack/.pwd
ontrack_production_db: geheim
```

Das ist auch kein hundertprozentiger Schutz, aber sicher besser, als Passwörter im Klartext über Konfigurationsdateien leicht zugänglich zu machen. Egal wie, gehen Sie bei der Nutzung der Passwörter auf Nummer sicher.

Passwörter werden per Default in die Log-Datei der Anwendung (z.B. *log/production.log*) geschrieben. Sie können dies vermeiden, indem Sie bestimmte Attribute wie das Passwort wie folgt filtern:

Listing 11.13: *app/controllers/application.rb*

```
class ApplicationController < ActionController::Base
  filter_parameter_logging :password
  ...
end
```

Im Log erscheint dann die Ausgabe:

Listing 11.14: *log/production.log*

```
...
Parameters: {..."username"=>"ontrack", "password"=>"[FILTERED]" }
```

11.7 Auslieferung

Mit *Capistrano* steht ein hervorragendes Tool für die vollautomatisierte Auslieferung (Deployment) und Release-Management von Rails-Applikationen zur Verfügung. Wir empfehlen die Verwendung von Capistrano von der ersten Auslieferung an, selbst in Applikationen, die auf nur einem einzigen Server ausgeführt werden. Die Verwendung von Capistrano beschreiben wir in Kapitel 12 ausführlich.

11.8 Wartung

Wenn Sie Ihre Anwendung erfolgreich in Produktion gebracht haben, gilt es ab jetzt, sie mit möglichst wenig Aufwand zu betreiben. Im Folgenden haben wir einige Tipps für die Wartung des Systems zusammengestellt.

Log-Dateien

Die Anwendung schreibt eine Reihe von Log-Dateien:

- Apache Logs (z.B. *apache.log* und *error.log*),
- Rails Logs (*log/production.log* und *log/fastcgi.crash.log*),
- Optionales Datenbank-Log.

Bei vielen Requests können die Log-Dateien schnell mehrere Megabytes groß werden. Dies kann zu Speicherplatz-Problemen führen, und je größer die Datei ist, umso unhandlicher wird der Umgang mit ihr und umso zeitintensiver die Suche während einer Fehleranalyse. Daher empfiehlt es sich, die Dateien täglich oder wöchentlich zu rollieren und die alten Dateien zu archivieren (vgl. Abschnitt 11.8).

Dies geschieht am besten durch ein entsprechendes Skript, das periodisch über die Cron-Tabelle gestartet wird. Logging-Bibliotheken, wie z.B. *log4r*, bieten ebenfalls das Rollieren von Dateien an. Da jedoch jeder Mongrel-Prozess seinen eigenen Logger verwendet, wird das Rollieren ggf. von mehreren Prozessen gleichzeitig gestartet, und es kommt zu Problemen.

Sessions aufräumen

Verwenden Sie den *ActiveRecordStore* (vgl. Abschnitt 11.4), werden die erzeugten Sessions nicht automatisch gelöscht. Schreiben Sie hierfür ein Skript, das Sie periodisch über die Cron-Tabelle starten, z.B.:

```
$ RAILS_ENV=production ./script/runner
'ActiveRecord::Base.connection.delete(
  "DELETE FROM sessions
  WHERE updated_at < now - 10*60*60")'
```

Wenn Sie eine Session nach einer Zeit der Inaktivität des Benutzers löschen möchten, hilft Ihnen dabei das Plugin *SessionTimeout*, das Sie unter http://opensource.agile-evolved.com/trac/browser/rails_plugins finden.

Backup

Selbstverständlich erstellen Sie regelmäßige Backups Ihres Systems. Vielleicht führt Ihr Provider bereits ein tägliches Backup durch, und das reicht Ihnen. Ist dies nicht der Fall, sind Sie gut beraten, ein eigenes periodisches Backup des gesamten Rechners auf einem anderen Rechner (z.B. per *rsync*) durchzuführen.

Die Anwendung ist selbst jederzeit aus dem Versionskontrollsystem von Anfang an zu installieren und muss nicht zwangsläufig mit gesichert werden. Sie sollten sich angewöhnen, Änderungen nicht direkt am produktiven System durchzuführen, da sonst der Produktionsstand und der Stand im Versionskontrollsystem auseinandergeraten. Bei der nächsten Einspielung fehlen dann ggf. Änderungen und müssen erneut durchgeführt werden. Wissen Sie dann noch, welche?

Auf jeden Fall sollten periodisch alle anwendungsrelevanten Daten, d.h. mindestens die Datenbank und alle Log-Dateien, auf einem anderen Rechner gesichert werden. Bei Bedarf wird so ein relativ aktueller Stand der produktiven Daten wiederhergestellt. Außerdem besteht jederzeit die Möglichkeit, auch Tage oder Wochen später aus den Log-Dateien und dem Datenbank-Stand Aktionen Ihrer Kunden nachzuvollziehen. Ebenso helfen Ihnen alte Log-Dateien zu analysieren, seit wann oder unter welchen Umständen technische Probleme auftraten. Ganz abgesehen von möglichen gesetzlichen Aufbewahrungsfristen sollten Sie alte Log-Dateien daher nicht einfach löschen.

Auch diese Aufgabe kann durch ein Skript täglich automatisch ausgeführt werden. In diesem Zusammenhang können Sie gleich die Log-Dateien rollieren. Vergessen Sie nicht, auch alle erstellten Skripte in einem Versionskontrollsystem oder über ein Backup zu sichern.

Console-Skript nutzen

Stellen Sie ein Problem in der Datenbank fest, kann dies ggf. über das *console*-Skript von Rails behoben werden (vgl. Abschnitt 4.3), sofern Sie keinen anderen Datenbank-Client nutzen.

11.9 Überwachung

Das Monitoring, d.h. die Überwachung der Anwendung, dient dazu, jederzeit den aktuellen Status der Anwendung zu erfahren. Dazu zählen auftretende Fehler oder die Auslastung. Stehen Ihre Anwendung oder Teile davon eine Zeit lang nicht zur Verfügung, schadet dies nicht nur Ihrem Image, sondern kann darüber hinaus nachhaltigen finanziellen Schaden bedeuten.

Eine der simpelsten Formen der Überwachung besteht darin, sich über auftretende Ausnahmen per E-Mail informieren zu lassen.

Installieren Sie dazu das Plugin *Exception_Notifier*, das Sie unter http://agilewebdevelopment.com/plugins/exception_notifier finden. Es sendet im Falle bestimmter Fehler eine E-Mail mit Kontextinformationen. Wir empfehlen Ihnen die Installation.

Ein etwas weiter gehender Schritt ist die Ausführung eines Test-Anwendungsfalls. Hierdurch testen Sie den Weg durch die Anwendung, d.h vom Apache über Mongrel über die Anwendung bis in die Datenbank. Lassen Sie diesen Test als Skript oder Ruby-Programm periodisch über die Cron-Tabelle laufen, und verschicken Sie im Fehlerfall eine E-Mail. Sie prüfen so ständig Ihr System und können im Bedarfsfall schnell reagieren.

Alternativ können Sie entsprechend professionelle Werkzeuge wie z.B. *Monit*¹⁴, *God*¹⁵ oder *Nagios*¹⁶ usw. installieren.

¹⁴<http://www.tildeslash.com/monit/>

¹⁵<http://god.rubyforge.org/>

¹⁶<http://www.nagios.org/>

Kapitel 12

Deployment mit Capistrano (und Subversion)

Deployment bezeichnet den Prozess der Auslieferung und Verteilung von Software (in unserem Fall einer Rails-Anwendung) auf die für deren Betrieb vorgesehenen Server. Darüber hinaus beinhaltet der Deployment-Prozess den Neustart der beteiligten Komponenten, wie Web- und Applikationsserver, sofern dies erforderlich ist.

Deployment ist kein einmaliger, sondern ein sich ständig wiederholender Prozess, der – sofern manuell ausgeführt – einen nicht unerheblichen Aufwand mit einer Reihe von Fallstricken darstellt. Diesen Umstand hat Jamis Buck frühzeitig erkannt. Jamis war es lästig, für jedes neue Deployment zunächst die aktuelle Version aus dem RCS (Revision Control System) auszuchecken, diese dann manuell auf die Betriebsserver zu kopieren, um anschließend die Webserver bzw. FastCGI-Prozesse von Hand neu zu starten.

Ziel von Jamis war die Entwicklung eines vollautomatisierten Deployment-Tools für Rails-Anwendungen, das die Auslieferung neuer Versionen und den Neustart der beteiligten Serverprozesse quasi auf Knopfdruck ermöglicht. Das Ergebnis dieser Arbeit ist ein Werkzeug namens *Capistrano* (ehemals SwitchTower), das das Deployment von Rails-Anwendungen zum Kinderspiel macht.

Capistrano hat sich mittlerweile zu „dem“ Standard-Werkzeug für die automatisierte Auslieferung von Rails-Anwendungen entwickelt. Dieses Kapitel führt zunächst anhand eines Quickstart-Tutorials in die Arbeitsweise und Verwendung von Capistrano ein. Darauf aufbauend, erläutern wir weitergehende Capistrano-Konzepte, wie die Entwicklung eigener oder die Erweiterung existierender Capistrano-Tasks.

12.1 Quickstart: Capistrano in 10 Minuten

Die Arbeitsweise von Capistrano lässt sich am einfachsten anhand eines Beispiels erläutern, in dem wir zeigen, wie die in Kapitel 3 entwickelte OnTrack-Anwendung per Capistrano auf einen Server deployed wird.

Abbildung 12.1 skizziert das Setup der den folgenden Ausführungen zugrunde liegenden Entwicklungs- und Deploymentumgebung.

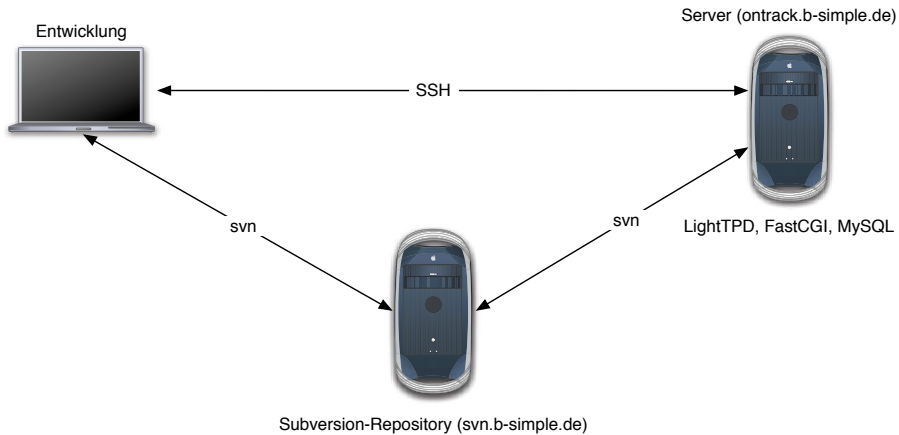


Abbildung 12.1: Eine einfache Produktionsumgebung

Die Entwicklung von OnTrack erfolgt auf dem Entwicklungsrechner. Dies ist der Rechner, auf dem Capistrano installiert ist und von dem Capistrano gestartet wird. Das Ziel des Deployments ist der Server *ontrack.b-simple.de*, der den Webserver LightTPD, die FastCGI-Prozesse und die Datenbank ausführt. Die Quellen der Anwendung liegen in einem Subversion-Repository auf dem Rechner *svn.b-simple.de*.

12.1.1 Voraussetzungen

Stellen Sie sicher, dass auf Ihrem Entwicklungsrechner folgende Komponenten installiert sind:

- Ruby 1.8.4 oder höher
- Rails 2.0 oder höher
- Rake 0.7.3 oder höher
- ssh-Client

Auf dem Server müssen folgende Komponenten vorhanden sein:

- Ruby 1.8.4 oder höher
- Rails 2.0 oder höher

- Rake 0.7.3 oder höher
- Subversion-Client (svn) bzw. alternativ CVS- oder Darcs-Client
- sshd-Prozess
- FastCGI
- ruby-fcgi
- LightTPD
- MySQL (oder eine andere Datenbank Ihrer Wahl)

Richten Sie auf dem Deploymentserver außerdem einen Benutzer ein, der sich per *ssh* auf dem Server anmelden kann, z.B.:

```
$ ssh ontrack.b-simple.de -l rails
```

Einige Hinweise zu Subversion

Wesentlicher Bestandteil eines Capistrano-Setups ist ein Revision Control System (RCS) zur Verwaltung der Sources. Capistrano arbeitet mit Subversion, CVS und Darcs zusammen. Die nachfolgenden Beschreibungen gehen von einem funktionierenden Subversion-Setup aus, dem von uns für Rails- und andere Projekte favorisierten RCS.

Die Beschreibung der Installation und Benutzung von Subversion würde den Rahmen dieses Kapitels bzw. Buches sprengen. Ein kostenloses Subversion-Handbuch finden Sie unter <http://svnbook.red-bean.com>. Die Subversion Software kann unter <http://subversion.tigris.org> für alle gängigen Betriebssysteme bezogen werden. Hinweise zum Einsatz von Rails zusammen mit Subversion finden Sie im Web unter <http://wiki.rubyonrails.com/rails/pages/HowtoUseRailsWithSubversion>.

Wir wollen Sie in Bezug auf die Installation von Subversion jedoch nicht völlig im Regen stehen lassen und listen im Folgenden grob die notwendigen Schritte zur Installation von Subversion auf den beteiligten Rechnern auf. Verstehen Sie die nachfolgenden Punkte bitte nicht als exakte Anleitung zur Installation von Subversion. Die Schritte sind mehr als grobe Hinweise gedacht, um Ihnen einen roten Faden vorzugeben. Lesen Sie die notwendigen Detail- und Hintergrundinformationen unbedingt unter <http://svnbook.red-bean.com> nach.

1. Installieren Sie auf Ihrem Entwicklungsrechner einen Subversion-Client (z.B. die Kommandozeilenvariante *svn*).
2. Installieren Sie auf Ihrem RCS-Server die Subversion-Serversoftware.
3. Richten Sie auf Ihrem RCS-Server ein Repository für OnTrack ein:

```
svn.b-simple.de:/home/bsimple> mkdir repos
svn.b-simple.de:/home/bsimple> cd repos
svn.b-simple.de:/home/bsimple/repos> svnadmin create ontrack
```


4. Starten Sie den Subversion-Serverprozess *svnserve* auf Ihrem RCS-Server:

```
svn.b-simple.de:/home/bsimple> svnserve --daemon \
                                --root /home/bsimple/repos
```

5. Ändern Sie auf dem RCS-Server die Default-Rechte anonymer Subversion-Benutzer von „Nur Lesen“ auf „Schreiben“:

Listing 12.1: /home/bsimple/repos/ontrack/conf/svnserve.conf

```
[general]
anon-access = write
```

6. Importieren Sie von Ihrem Entwicklungsrechner aus die aktuellen OnTrack-Quellen ins RCS:

```
$ cd ontrack
$ svn import . -m "initial import" \
  svn://svn.b-simple.de/ontrack/trunk
```

7. Ersetzen Sie die lokale Kopie von OnTrack auf Ihrem Entwicklungsrechner durch die Version aus dem RCS:

```
$ rm -rf ontrack
$ svn co svn://svn.b-simple.de/ontrack/trunk ontrack/trunk
```

8. Installieren Sie auf dem Deploymentserver den Subversion-Client *svn*.

12.1.2 Installation von Capistrano

Bevor es losgehen kann, muss Capistrano auf dem Entwicklungsrechner installiert werden:

```
$ sudo gem install capistrano --include-dependencies
```

Der Gem-Aufruf installiert neben Capistrano eine Reihe von zusätzlich benötigten Gems, wie das Dependency-Injection Framework *needle* oder das für die SSH-basierte Kommunikation benötigte Paket *net-ssh*.

12.1.3 Anwendung Capistrano-ready machen

Nach der Installation muss die Anwendung für die Verwendung von Capistrano vorbereitet werden. Wechseln Sie dafür in das Hauptverzeichnis der Anwendung und rufen dort das Kommando *capify* gefolgt von einem Punkt auf:

```
$ cd ontrack/trunk
$ capify .
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

Das Kommando erzeugt zwei Konfigurationsdateien: das *Capfile* und die Datei *deploy.rb* im *config*-Verzeichnis. Das *Capfile* ist eine Art Makefile für Capistrano und enthält Konfigurationseinstellung und Kommandos zur Ausführung von Capistrano-Tasks.

In Rails-Anwendungen gilt die Konvention, dass Konfigurationsdateien unterhalb des *config*-Verzeichnisses liegen müssen. Aus diesem Grunde erfolgt die Capistrano-Konfiguration bei der Verwendung innerhalb von Rails nicht im *Capfile* direkt, sondern in der Datei *config/deploy.rb*. Die Datei *deploy.rb* wird standardmäßig von *Capfile* eingebunden, so dass wir uns von hier an ausschließlich auf *deploy.rb* als zentrale Konfigurationsdatei konzentrieren werden. Die Konfigurationsdatei bezeichnen wir im Folgenden als *Deploymentrezept* der Anwendung.

12.1.4 Konfiguration

Die Anwendung ist jetzt „Capistrano-ready“ und muss als Nächstes konfiguriert werden. Capistrano unterscheidet drei Konfigurationselemente:

- **Variablen.** Steuern die Arbeitsweise von Capistrano, indem ihnen mit *set* ein Wert zugewiesen wird.
- **Rollen.** Klassifizieren die verschiedenen Server der Deploymentumgebung.
- **Tasks.** Führen die eigentlichen Deploymentschritte durch.

Alle Elemente werden in der Konfigurationsdatei *deploy.rb* gesetzt bzw. definiert. Im Folgenden werden wir die von Ihnen zu setzenden Konfigurationsoptionen der Reihe nach durchgehen. Die vollständige Konfigurationsdatei finden Sie noch einmal zur Übersicht ganz hinten in diesem Kapitel unter Abschnitt 12.9.

Applikationsname

Capistrano erwartet den Namen der Anwendung unter der Variablen *application*:

```
set :application, "ontrack"
```

Die Variable wurde bereits mit dem Wert *ontrack* initialisiert, sofern Sie den Anwendungsnamen beim Erzeugen von *deploy.rb* übergeben haben (siehe Abschnitt 12.1.3).

Subversion-Repository

Das Subversion-Repository für *OnTrack* befindet sich auf dem Server *svn.b-simple.de* im Unterverzeichnis *ontrack/trunk* und wird in *deploy.rb* wie folgt konfiguriert:

```
set :repository, "svn://svn.b-simple.de/#{application}/trunk"
```

Rollen

Anschließend erfolgt die Vergabe von Rollen. Capistrano ordnet den verschiedenen Servern der Deploymentumgebung drei unterschiedliche Rollen zu:

- **:web.** Webserver der Umgebung, d.h. ein Server, auf dem der HTTP-Prozess (z.B. Apache oder LightTPD) läuft.
- **:app.** Applikationsserver der Umgebung, d.h. ein Server, auf dem die FastCGI-Prozesse und der Ruby-Code der Anwendung laufen.
- **:db.** Datenbankserver der Umgebung.

In unserem Beispielsetup werden alle drei Rollen von ein und demselben Server übernommen: *ontrack.b-simple.de*. Entsprechend sehen die Einträge im Deploymentrezept aus:

```
role :web, "ontrack.b-simple.de"
role :app, "ontrack.b-simple.de"
role :db, "ontrack.b-simple.de", :primary => true
```

Neben Rollen-Bezeichnung und Servernamen akzeptiert *role* als dritten Parameter eine optionale Hash. Die Rolle *db* macht hiervon Gebrauch und spezifiziert mit Hilfe von *:primary => true*, dass es sich bei dem Datenbankserver um den primären Server handelt. Diese zusätzliche Information ermöglicht das Programmieren von Tasks, die nur auf bestimmten Servern einer Rolle ausgeführt werden. Ein Beispiel hierfür ist der Task *remote:migrate*, der nur auf primären Datenbankservern ausgeführt wird (siehe 12.2).

Im Folgenden sprechen wir von „Server“, wenn wir keine spezielle Serverrolle bezeichnen.

Deployment-Ziel

Die Variable *deploy_to* gibt den Zielort des Deployments an, d.h. das Verzeichnis auf dem Applikationsserver (Rolle *app*), in das die Sourcen ausgecheckt werden:

```
set :deploy_to, "/home/rails/#{application}"
```

Deployment-Benutzer

Capistrano wird auf dem lokalen Entwicklungsrechner gestartet und verbindet sich via SSH auf den Server. Standardmäßig verwendet Capistrano für die SSH-Verbindung das Login des lokal angemeldeten Benutzers. Ist dieser Benutzer auf dem Server nicht vorhanden, kann der für das Deployment zu verwendende Benutzer explizit über die Variable *:user* vorgegeben werden:

```
set :user, "rails"
```

FastCGI-Prozesse ohne sudo neu starten

Im Zuge eines Deployments startet Capistrano die FastCGI-Prozesse der Anwendung neu. Standardmäßig erfolgt dies über das Kommando *sudo*, das die Prozesse unter dem Benutzer *root* neu startet. Unser Setup basiert darauf, dass nur der LightTPD unter *root* läuft, die zugehörigen FastCGI-Prozesse aber unter dem „normalen“ Benutzer *rails* ausgeführt werden.

Der Neustart der FastCGI-Prozesse erfordert damit keine Root-Berechtigung. Entsprechend sollte die Verwendung von *sudo* im Deploymentrezept deaktiviert werden:

```
set :use_sudo, false
```

12.1.5 Setup des entfernten Verzeichnisses

Vor dem ersten Deployment muss eine Capistrano-konforme Verzeichnisstruktur auf den Servern vorhanden sein. Erstellen Sie diese Struktur durch Aufruf des Capistrano-Tasks *deploy:setup*:

```
$ cap deploy:setup
```

Der Aufruf erzeugt die folgende Verzeichnisstruktur auf dem Server *ontrack.b-simple.de*:

```
home
  rails
    ontrack
      releases
      shared
```

Im Verzeichnis *releases* legt Capistrano für jedes neue Deployment ein Unterverzeichnis (Name = aktuelles Datum + Uhrzeit nach UTC) an, in das die jeweils neuesten Sourcen der Anwendung ausgecheckt werden. Das Verzeichnis *shared* dient als Ablage für Dateien bzw. Verzeichnisse, die für alle Deployments gleich sind. Mehr dazu finden Sie in Abschnitt 12.7.

12.1.6 Erstes Deployment

Die Anwendung ist damit für ein Capistrano-Deployment konfiguriert und kann das erste Mal deployed werden. Der Unterschied zwischen dem ersten und allen folgenden Deployments besteht darin, dass wir vor dem ersten Deployment den HTTP-Server und die zugehörigen FastCGI-Prozesse starten müssen, während bei allen folgenden Deployments nur die FastCGI-Prozesse neu zu starten sind.

Spawner: FastCGI-Prozesse starten

Unser Setup verwendet LightTPD mit externen FastCGI-Prozessen. Der Initialstart eines oder mehrerer FastCGI-Prozesse erfolgt über das Skript *spawner*. Der Spawner-Aufruf wird in das Deploymentrezept eingebunden, indem der Task *deploy:start* wie folgt überschrieben wird:

```
namespace :deploy do
  desc "Spawn FastCGI-Listener"
  task :start do
    run "nohup #{current_path}/script/process/spawner -p 8000 -i 1"
    run "cat /home/rails/nohup.out"
    run "rm /home/rails/nohup.out"
  end
end
```

Der Parameter *-p* gibt den initialen Port und der Parameter *-i* die Anzahl der zu startenden FastCGI-Instanzen an. Wird mehr als eine Instanz angegeben, dann wird die initiale Portnummer in 1-er Schritten hochgezählt, d.h. im Beispiel würde Prozess 1 die Portnummer 8000 bekommen, Prozess 2 die Portnummer 8002, usw. Achten Sie darauf, dass die hier spezifizierte(n) Portnummer(n) mit denen der LightTPD-Konfigurationsdatei unter der Option *fastcgi.server* angegebene(n) Portnummer(n) übereinstimmen müssen (siehe Abschnitt 12.1.6).

Das dem Spawner-Aufruf vorangestellte Kommando *nohup* ist unserer Erfahrung nach zwingend erforderlich, da es ohne *nohup* zu SSH-Fehlern kommt und die FastCGI-Prozesse nicht gestartet werden. Das Problem liegt vermutlich darin begründet, dass Capistrano die SSH-Verbindung zum Server beendet, bevor das Spawner-Skript vollständig ausgeführt wurde.

Die Verwendung von *nohup* behebt dieses Problem, allerdings zu Lasten fehlender Erfolgs- oder Fehlermeldungen. Die Verwendung von *nohup* lenkt sämtliche Ausgaben des Spawners in die Datei */home/rails/nohup.out* um, sodass Sie keine Rückmeldung darüber erhalten, ob die FastCGI-Prozesse erfolgreich gestartet wurden. Für Abhilfe sorgen hier die beiden *run*-Anweisungen am Ende des Spinner-Tasks, die den Inhalt von *nohup.out* ausgeben und die Datei anschließend löschen.

Anwendung deployen

Die Vorbereitungen sind abgeschlossen, und die Anwendung kann das erste Mal deployed werden:

```
$ cap deploy:cold
```

Das Target *deploy:cold* meldet sich per SSH auf dem Server an und checkt dort die neueste OnTrack-Version in ein neues Unterverzeichnis (Name = Aktuelles Datum und Uhrzeit) von */home/rails/ontrack/releases* aus. Sofern der konfigurierte SSH-Benutzer nicht über SSH-Schlüssel authentifiziert werden kann, werden Sie nach dem Passwort des Benutzers gefragt.

Nach dem Auschecken legt Capistrano einen Link */home/rails/ontrack/current* auf die neueste Version von OnTrack an. Anschließend wird der Task *deploy:start* ausgeführt, der einen FastCGI-Prozess auf Port 8000 startet.

LightTPD konfigurieren und starten

Was noch fehlt, sind die Konfiguration und der Start des HTTP-Servers LightTPD. Wir beginnen mit der Konfiguration von LightTPD. Melden Sie sich als *root* auf dem

Server an, und erstellen Sie im Verzeichnis `/etc/lighttpd` die Datei `lighttpd.ontrack` mit folgendem Inhalt:

Listing 12.2: `/etc/lighttpd/lighttpd.ontrack`

```
var.appname = "ontrack"
server.modules = ("mod_rewrite", "mod_accesslog", "mod_fastcgi")
server.port = 4000
server.username = "rails"
server.groupname = "www"
server.pid-file = "/var/run/lighttpd.pid"
accesslog.filename = "/tmp/access_log"
server.errorlog = "/tmp/error_log"
server.indexfiles = ( "index.html" )
url.rewrite = ( "^/$" => "index.html", "^([^.]+)$" => "$1.html" )
server.document-root = "/home/rails/" + var.appname + "/current/public"
server.error-handler-404 = "/dispatch.fcgi"

mimetype.assign = (
    ".css" => "text/css",
    ".gif" => "image/gif",
    ".html" => "text/html",
    ".jpeg" => "image/jpeg",
    ".jpg" => "image/jpeg",
    ".js" => "text/javascript",
    ".pdf" => "application/pdf",
    ".png" => "image/png",
    ".txt" => "text/plain",
)

fastcgi.server = ( ".fcgi" => (
    "host"      => "127.0.0.1",
    "port"      => 8000,
)))
```

Abschließend kann der LightTPD gestartet werden (wieder unter dem Benutzer `root`):

```
ontrack.b-simple.de:> lighttpd -f /etc/lighttpd/lighttpd.ontrack
```

Wenn Sie bis jetzt alles richtig gemacht haben und auch sonst kein Fehler vorliegt,¹ können Sie Ihr erstes Deployment testen und sich über den Browser auf OnTrack verbinden: `http://ihre-server-adresses:4000/project/login`. Allerdings wird jeder Anmeldeversuch noch fehlschlagen, da die Anwendung bisher noch ohne Datenbank deployed wurde. Wie das geht, erfahren Sie in Abschnitt 12.2.

¹ Hiervon gibt es bekanntlich viele: Bibliotheken liegen in der falschen Version vor, Benutzer haben nicht genügend Rechte, etc.

12.1.7 Fallstricke

Leider läuft nicht immer alles nach Plan. Dieser Abschnitt zählt die Probleme auf, die wir bei unseren bisherigen Capistrano-Deployments hatten.

Subversion-Client fehlt oder ist nicht im Pfad

Der Subversion-Client ist nicht installiert oder befindet sich nicht im Ausführungspfad des Deploymentsservers.

Lösung: Installieren Sie den Subversion-Client. Der zu verwendende Client kann über die Variable `:svn` explizit vorgegeben werden:

```
set :svn, "/usr/local/bin/svn"
```

Cold-Deploy findet spawn-fcgi nicht

Cold-Deploy startet FastCGI-Prozesse auf dem Server per ssh-basierter Ausführung von *spawn-fcgi*. SSH verwendet einen fest eincompilierten Ausführungspfad, so dass *spawn-fcgi* auf bestimmten Systemen nicht gefunden wird, z.B. dann, wenn *spawn-fcgi* in `/usr/local/bin` liegt.

Lösung: Erzeugen Sie auf dem Server einen Symlink im Verzeichnis `/usr/bin`, der auf `/usr/local/bin/spawn-fcgi` zeigt:

```
$ ssh ontrack.b-simple.de -l root
ontrack.b-simple.de:> cd /usr/bin
ontrack.b-simple.de:> ln -s /usr/local/bin/spawn-fcgi spawn-fcgi
```

Ausführung von Reaper und Spawner schlägt fehl

Die im Rahmen des initialen Deployments ausgeführten Tasks *restart* und *start* verwenden die beiden Skripte *reaper* und *spawner*. Beide Skripte verwenden das Kommando *env*, um den Ruby-Interpreter in der Umgebungsvariable *PATH* des Servers zu suchen:

```
#!/usr/bin/env ruby
```

Bei einer Verbindung über SSH wird der *PATH* vom SSH-Dämon neu gesetzt. Da dieser Pfad fest in den SSH-Dämon hineincompiliert wurde, ist das Verzeichnis mit dem Ruby-Interpreter möglicherweise nicht enthalten. Dies ist z.B. auf unseren Servern der Fall, auf denen *ruby* in `/usr/local/bin` liegt, *sshd* aber nur mit dem Pfad `/usr/bin:/bin:/usr/sbin:/sbin` kompiliert wurde.

Lösung: Erzeugen Sie auf dem Server einen Symlink im Verzeichnis `/usr/bin`, der auf den Ruby-Interpreter aus `/usr/local/bin` zeigt:

```
$ ssh ontrack.b-simple.de -l root
ontrack.b-simple.de:> cd /usr/bin
ontrack.b-simple.de:> ln -s /usr/local/bin/ruby ruby
```

12.1.8 Nachfolgende Deployments

Die auf das initiale Cold-Deployment folgenden Deployments sind denkbar einfach. Nachdem LightTPD und die zugehörigen FastCGI-Prozesse einmal gestartet wurden, reicht der folgende rake-Aufruf aus, um die neueste Version der Anwendung zu deployen und die FastCGI-Prozesse neu zu starten:

```
cap deploy
```

Der Task *deploy* verzichtet auf die Ausführung des von *deploy:cold* ausgeführten Start-Tasks, da die FastCGI-Prozesse ja bereits laufen und deshalb lediglich neu gestartet werden müssen.

12.2 Datenbanksetup und Migration

Damit OnTrack in der Deploymentumgebung funktioniert, wird auf dem Deploymentserver eine Produktionsdatenbank benötigt. Unter MySQL sieht der zugehörige Aufruf wie folgt aus:

```
$ ssh ontrack.b-simple.de
ontrack.b-simple.de:> mysql -u rails -p
Enter password: *****

mysql> create database ontrack_production;
```

Achten Sie darauf, dass der Eintrag *production* in der Datenbankkonfiguration *database.yml* korrekt konfiguriert ist:

Listing 12.3: config/database.yml

```
production:
  adapter: mysql
  database: ontrack_production
  host: localhost
  username: rails
  password: secret
```

Holen Sie die Konfiguration in der lokalen Arbeitskopie von *database.yml* nach, checken Sie die Datei ein, und deployen Sie die Software erneut (*cap deploy*).

Anschließend kann die Datenbank via Capistrano auf den neuesten Stand gebracht werden:

```
cap deploy:migrate
```

Das Kommando führt eine serverseitige Datenbankmigration mittels *rake* aus. Wird *rake* auf dem Server nicht gefunden, da es z.B. unter */usr/local/bin* liegt, kann das Problem durch Erzeugung eines Symlinks und erneute Ausführung von *cap deploy:migrate* behoben werden:

```
$ ssh ontrack.b-simple.de -l root
ontrack.b-simple.de:> cd /usr/bin
```



```
ontrack.b-simple.de:> ln -s /usr/local/bin/rake rake
ontrack.b-simple.de:> logout
$ cap deploy:migrate
```

Zur Erinnerung: Die Datenbankmigration wird nur auf primären Datenbankservern ausgeführt:

```
role :db, "ontrack.b-simple.de", :primary => true
```

Für zukünftige Deployments, die neben Änderungen im Sourcecode auch Datenbankänderungen beinhalten, wird statt *deploy* das Target *deploy:migrations* ausgeführt.

12.3 Rollback eines Release

Stellt sich heraus, dass eine neu deployte Version nicht so arbeitet, wie erwartet, kann die Version einfach zurückgerollt werden:

```
cap deploy:rollback
```

Der Task verlinkt das Verzeichnis *current* auf die Vorgängerversion, löscht die neueste Version aus dem Verzeichnis *releases* und startet die FastCGI-Prozesse neu.

12.3.1 Rollback mit Datenbankmigration

Das Deployment einer neuen Version kann neben neuen Sourcen auch eine neue Version des Datenbankschemas beinhalten. In einem solchen Fall muss ein Rollback zusätzlich eine Migration des Schemas auf dessen vorletzte Version beinhalten.

Der für die Datenbankmigration zuständige *migrate*-Task kennt keine Versionsnummern. Allerdings lässt sich das Verhalten von *migrate* über die Variable *:migrate_env* steuern. Folgender Eintrag in der Datei *deploy.rb* setzt *:migrate_env* auf eine per Kommandozeile übergebene Versionsnummer:

```
set :migrate_env, ARGV[1] if ARGV[1] =~ /VERSION=
```

Anschließend kann der Task *deploy:migrate* mit einer Versionsnummer aufgerufen und das Datenbankschema auf eine bestimmte Version zurückmigriert werden:

```
$ cap deploy:migrate VERSION=27
```

12.4 Tasks

Im Kern besteht Capistrano aus einer Reihe von Tasks für Deployment-Aufgaben. Capistrano liefert die wichtigsten und in vielen Fällen völlig ausreichenden Standardtasks mit (z.B. *remote:deploy*), ist jedoch offen für Erweiterungen, wie z.B. die Erstellung eigener Tasks oder das Überschreiben von Standardtasks. Das Kommando *cap -T* liefert eine Liste der verfügbaren Capistrano-Tasks:

```
$ cap -T
cap deploy                # Deploys your project.
cap deploy:check          # Test deployment dependencies.
cap deploy:cleanup        # Clean up old releases.
cap deploy:cold           # Deploys and starts a 'cold' application.
cap deploy:migrate        # Run the migrate rake task.
...
```

Tasks können nach Namespaces gruppiert werden. Ein Namespace definiert einen Namensraum für eine Gruppe zusammengehöriger Tasks. Beispielsweise sind sämtliche Deployment-Tasks im Namespace *deploy* zusammengefasst.

Neue Tasks werden im Deploymentrezept *deploy.rb* der Rails-Anwendung definiert. Ein Task besteht aus einem optionalen Namespace, einer optionalen Beschreibung *desc* und der eigentlichen Taskdefinition, bestehend aus dem Schlüsselwort *task*, gefolgt von einem Namen und einem Block:

```
namespace :experimental do
  desc "My Task"
  task :my_task do
    ...
  end
end
```

Der Block enthält die eigentlichen Arbeitsanweisungen des Tasks. Dies kann beliebiger Ruby-Code sein, beschränkt sich aber i.d.R. auf Task-Helper-Aufrufe, von denen Capistrano insgesamt sieben zur Verfügung stellt. Die beiden wichtigsten Task-Helper sind *run* und *sudo*:

- *run(command, options={}, &block)*
- *sudo(command, options={}, &block)*

Beide Methoden führen ein Kommando auf dem Server aus, wobei *run* den Befehl mit den Rechten des angemeldeten normalen (SSH) Benutzers und *sudo* den Befehl als root ausführt:

```
namespace :experimental do
  task :my_task do
    run "echo Hostname is $HOSTNAME"
  end
end
```

Der Task führt das Unix-Kommando *echo* auf dem Server aus und überträgt die Ausgaben des Kommandos zurück zum Client (dem Entwicklungsrechner), wo sie auf der Shell ausgegeben werden.

12.4.1 Ausführen von Tasks

Selbstdefinierte Capistrano-Tasks werden über *cap* gefolgt von Namespace und Tasknamen ausgeführt:

```
$ cap experimental:my_task
```

Standardmäßig führt Capistrano einen Task parallel auf allen konfigurierten Zielmaschinen (Rollen) aus. Eine dedizierte Angabe der Zielrechner ist über den Parameter *:roles* möglich:

```
namespace :experimental do
  task :my_task, :roles => [:web, :app] do
    run "echo Hostname is $HOSTNAME"
  end
end
```

Für Datenbankserver kann zusätzlich über den Parameter *:only* gesteuert werden, ob der Task nur auf dem primären oder auf allen konfigurierten Datenbankservern ausgeführt werden soll:

```
task :backup, :roles => :db, :only => { :primary => true }
...
end
```

Taskausführung über *invoke* und *shell*

Alternativ zur Definition neuer Tasks können einzelne Unix-Kommandos auch direkt, d.h. mit Hilfe des *invoke*-Tasks ausgeführt werden. Das folgende Beispiel liefert den zur Verfügung stehenden Plattenplatz auf Ihren Deploymentservern:

```
cap invoke COMMAND="df -h"
```

Neben *invoke* enthält Capistrano den *shell*-Task für die direkte Kommandoausführung auf entfernten Servern:

```
cap shell
```

Einmal gestartet, können Sie jetzt beliebige Unix-Kommandos direkt eingeben:

```
cap> df -h
...
cap> echo $HOSTNAME
...
```

Standardmäßig führt die Shell das eingegebene Kommando auf allen im Deploymentrezept konfigurierten Servern aus. Mit Hilfe von *on* (für Servernamen) und *with* (für Rollennamen) kann die Serverauswahl explizit angegeben werden:

```
cap> on bsimple df -h
...
cap> with db echo $HOSTNAME
...
```

Während *invoke* für jedes Kommando eine SSH-Verbindung aufbaut, kommt *shell* mit einer Verbindung pro Server aus. Wollen Sie also mehr als ein Kommando absetzen oder auf Ihren Servern mit verschiedenen Kommandos experimentieren, dann ist *shell* der Task Ihrer Wahl.

12.4.2 Mehr über run und sudo

Prinzipiell kann über *run* oder *sudo* jedes beliebige POSIX-konforme Shell-Kommando auf dem Server ausgeführt werden. Beispiele sind *rm* zum Löschen von Dateien und Verzeichnissen oder *ln* zum Anlegen von Links.

Ausgaben des entfernten ausgeführten Shell-Kommandos werden zurück zum Client (der Rechner, auf dem der cap-Aufruf ausgeführt wird) transportiert und dort auf der Shell ausgegeben:

```
** [out :: ontrack.b-simple.de] Hostname is ontrack.b-simple.de
```

Ein an *run* oder *sudo* übergebener Block wird für jede Ausgabe des entfernten Kommandos aufgerufen und muss drei Parameter akzeptieren:

1. **channel.** SSH-Kanal für die Kommunikation mit dem Server.
2. **stream.** Stream-Identifizier (stdout oder stderr).
3. **:data.** Aktuelle Ausgabe des entfernten Kommandos.

Ein gutes Beispiel für die Benutzung von Blöcken ist die Ausführung von entfernten Kommandos, die die Eingabe des Passworts erfordern. Das folgende Beispiel verwendet statt des Capistrano-sudo den Befehl *run*, führt dafür aber serverseitig einen *sudo*-Aufruf durch, der den Benutzer nach dem Root-Passwort fragt:

```
task :clear_sessions do
  run "sudo rm /tmp/ruby_sess.*" do |channel, stream, data|
    if data =~ /^Password:/
      logger.info "#{channel[:host]} asked for root password"
      pw = Capistrano::CLI.password_prompt("Root Password: ")
      channel.send_data "#{pw}\n"
    end
  end
end
```

Enthält ein einzelner *run* oder *sudo*-Aufruf mehr als ein Shell-Kommando, sind die einzelnen Kommandos durch *&&* zu trennen:

```
task :rollback do
  run <<-CMD
    ln -nfs #{previous_release} #{current_path} &&
    rm -rf #{current_release}
  CMD
end
```

Tasks können andere Tasks durch Angabe von Namen aufrufen:

```
task :show_hostname do
  my_task
end
```

12.4.3 Weitere Task-Helper: delete, put und render

Neben *run* und *sudo* stehen drei weitere Task-Helper zur Verfügung:

- **delete(path, options={}).** Löscht die Datei oder das Verzeichnis auf allen Servern. Über den Parameter *:recursive* wird gesteuert, ob Verzeichnisse rekursiv gelöscht werden, d.h. ob *rm -rf* statt *rm -f* verwendet wird:

```
delete "/home/rails/ontrack/current", :recursive => true
```

- **put(buffer, path, options={}).** Erzeugt oder überschreibt die Datei auf allen Servern und füllt sie mit dem Inhalt von *buffer*. Der optionale Parameter *:mode* dient der Angabe der Dateirechte, z.B.:

```
put buffer, "#{shared_path}/system/maintenance.html",
:mode => 0644
```

- **render(template, options={}) bzw. render(options={}).** Rendert das angegebene Template und liefert einen String. Render ist damit weniger ein „Server“-Kommando, sondern vielmehr eine einfache Methode zum Erzeugen von ERB-Templates. Der Bezug zu Capistrano entsteht dann, wenn man die von render erzeugten Templates per *put* auf den Server überträgt, um sie dort z.B. als Wartungsseite zu hinterlegen:

```
buffer = render("maintenance.rhtml", :deadline => ENV['UNTIL'])
put buffer, "#{shared_path}/system/maintenance.html",
:mode => 0644
```

12.4.4 Transaktionen und Rollbacks

Bleiben noch die letzten beiden zu erwähnenden Task-Helper: *transaction* und *on_rollback*. Die Helper ermöglichen die Ausführung mehrerer Task-Kommandos innerhalb einer Transaktion. Die Transaktion wird über den Helper *transaction* definiert, der die auszuführenden Befehle als Block erwartet:

```
namespace :deploy do
  task :my_deploy do
    transaction do
      update_code
      disable_web
      ...
    end
  end
end
```

Schlägt einer der Tasks innerhalb der Transaktion fehl, werden die bis dahin bereits ausgeführten Tasks rückgängig gemacht, sofern sie über einen Rollback-Handler verfügen. Rollback-Handler werden innerhalb eines Task-Blocks über den Helper *on_rollback* definiert:

```
namespace :deploy do
  task :update_code do
    on_rollback { delete release_path, :recursive => true }
    ...
  end
end
```

Im Beispiel führt *my_deploy* den Task *update_code* innerhalb einer Transaktion aus. Schlägt z.B. der im Anschluss an *update_code* ausgeführte Task *disable_web* fehl, rollt Capistrano die Transaktion zurück. Der Rollback-Handler von *update_code* wird ausgeführt, wodurch die zuvor ausgecheckten Sourcen wieder gelöscht werden.

12.4.5 Überschreiben von Standardtasks

Standardtasks werden überschrieben, indem der Task im Deploymentrezept unter demselben Namen redefiniert wird. Ein Beispiel hierfür ist die Redefinition des *start*-Tasks aus Abschnitt 12.1.6:

```
namespace :deploy do
  desc "Spawn FastCGI-Listener"
  task :start do
    run "nohup #{current_path}/script/process/spawner -p 8000 -i 1"
  end
end
```

Für überschriebene Tasks gilt, dass Capistrano immer die Neudefinition des Tasks ausführt und die ursprüngliche Version ignoriert.

Hinweis: Das Überschreiben von Tasks funktioniert nur, wenn dieselben Namensräume wie in der Definition des Ursprungstasks verwendet werden. Bezogen auf den *start*-Task bedeutet dies, dass der Task innerhalb des Namensraums *deploy* redefiniert werden muss.

12.4.6 Tasks erweitern: Before- und After-Tasks

Einige Tasks erfordern Vor- oder Nacharbeiten. Ein Beispiel ist das Erzeugen von Dateisystem-Links nach dem Auschecken der aktuellen Version des Systems (vgl. Abschnitt 12.7.1).

Tasks werden durch so genannte Before- bzw. After-Tasks erweitert:

```
task :before_my_task do
  puts "before 'my_task'"
end

task :my_task do
  run "echo Hostname is $HOSTNAME"
end

task :after_my_task do
```

```
puts "after 'my_task' "
end
```

Ein Task, dessen Name mit *before_* oder *after_* beginnt, wird automatisch vor bzw. nach Ausführung des eigentlichen Tasks (im obigen Beispiel *my_task*) ausgeführt.

Task-Erweiterungen können eigene Rollen besitzen, wodurch es z.B. möglich wird, *my_task* auf allen Servern, *before_my_task* aber nur auf der Datenbank auszuführen. Ein Anwendungsfall ist z.B. die Definition eines Tasks *before_deploy*, der vor dem Deployment eine Sicherung der aktuellen Datenbank durchführt:

```
task :before_deploy, :roles => :db do
  ...
end
```

Die Angabe des Parameters *:roles* stellt sicher, dass die Datenbank nur auf dem Datenbankserver gesichert wird.

12.5 Variablen

Variablen sind neben Rollen und Tasks die dritte Komponente einer Capistrano-Konfiguration. Capistrano verfügt über eine Reihe vordefinierter Variablen, die die Arbeitsweise von Capistrano steuern. Vier von ihnen haben Sie bereits in Abschnitt 12.1 kennengelernt: *:application*, *:repository*, *:deploy_to* und *:user*.

Variablen wird über *set* ein Wert zugewiesen, z.B.:

```
set :user, "rails"
```

Die nachfolgende Tabelle beschreibt die vordefinierten Capistrano-Variablen:

Variable	Default	Beschreibung
application	erforderlich	Name der Applikation
checkout	co	Subversion-Kommando zum Checkout von Sourcen
cvs	PATH	Verzeichnis, in dem der cvs-Client auf dem entfernten Server installiert ist
current_dir	current	Name des Links unterhalb von <i>deploy_to</i> , der auf die jeweils aktuelle Version zeigt
current_path	von Capistrano gesetzt	Absoluter Link auf die jeweils aktuelle Version der Anwendung
darcs	PATH	Verzeichnis, in dem der darcs-Client auf dem entfernten Server installiert ist
deploy_to	/u/apps/#{application}	Rootverzeichnis des Deployments
gateway	optional	Adresse eines Gatewayservers, alle Verbindungen werden über diesen Server getunnelt.

password	Eingabeaufforderung	Passwort für das Login auf den Servern
repository	erforderlich	URL des RCS Repositories
release_path	von Capistrano gesetzt	Absolutes Verzeichnis, in das die Anwendung beim jeweiligen Deployment installiert wird
revision	RCS abhängiges Format	Revisionsnummer der auszucheckenden Version
scm	:subversion	Zu verwendendes RCS (:subversion, :cvs oder :darcs)
shared_dir	shared	Name des „shared“-Verzeichnisses unterhalb von <i>deploy_to</i>
ssh_options	Hash.new	Hash mit abweichenden SSH-Optionen, z.B. zum Überschreiben des Standardports: <code>ssh_options[:port] = 1234</code>
svn	PATH	Verzeichnis, in dem der svn-Client auf dem entfernten Server installiert ist
user	angemeldeter Benutzer	Benutzername für das Login auf den Servern
use_sudo	true	Gibt an, ob Sudo-Tasks mittels <i>sudo</i> ausgeführt werden oder ob alternativ <i>run</i> verwendet wird
version_dir	releases	Verzeichnis für die einzelnen Releases unterhalb von <i>deploy_to</i>

12.6 FastCGI-Utilities

Rails erzeugt für jede Applikation zwei Skripte zum Start und Neustart von FastCGI-Prozessen: *Spawner* und *Reaper*.

12.6.1 Spawner

Das Spawner-Skript liegt unter *ontrack/script/process/spawner* und dient dem initialen Start von FastCGI-Prozessen. Wir haben den Spawner bereits im Rahmen der Redefinition des Spinner-Tasks kennengelernt, der den Spawner zum Start eines FastCGI-Prozesses auf Port 8000 nutzt (siehe Abschnitt 12.1.6).

Der Spawner verwendet zum Starten der FastCGI-Prozesse das Unix-Kommando *spawn-fcgi*. Stellen Sie sicher, dass sich dieses Programm im SSH-Ausführungspfad des Servers befindet.

12.6.2 Reaper

Das Reaper-Skript liegt unter *ontrack/script/process/reaper* und startet laufende FastCGI-Prozesse neu. Der Neustart erfolgt „gracefully“, d.h. die in Arbeit befindlichen HTTP-Requests werden vor dem Neustart vollständig abgearbeitet.

Das Reaper-Skript wird vom Standardtask *restart* verwendet. Der Reaper versucht, die neu zu startenden FastCGI-Prozesse über deren Namen ausfindig zu machen (via *grep*). In unserem Setup deployen wir die Anwendung nach */home/rails/ontrack* (siehe 12.1.4). Der Reaper erweitert diesen Pfad um *current/public/dispatch.fcgi* und sendet allen Prozessen, die das Programm */home/rails/ontrack/current/public/dispatch.fcgi* ausführen, ein USR2-Signal.

12.7 Gemeinsame Dateien – das Shared-Verzeichnis

Für einige Dateien ist es sinnvoll und notwendig, dass sie einzelne Deployments überdauern, d.h. von Deployment zu Deployment in derselben Version zur Verfügung stehen, ohne dass sie aus dem RCS ausgecheckt werden müssen. Beispiele hierfür sind Log-Dateien, Datenbank-Konfigurationsdateien oder auch Anwendungsdaten, wie generierte PDFs.

Für diesen Zweck gibt es auf dem Server ein Verzeichnis *shared*, in dem sich Dateien befinden, die über einzelne Deployments hinweg weder gelöscht noch durch neue Versionen überschrieben werden. Das Verzeichnis wird vom Capistrano-Task *deploy:setup* angelegt und befindet sich auf derselben Ebene wie das *releases*-Verzeichnis:

```
home
  rails
    ontrack
      releases
      shared
      log
```

Capistrano verwendet das *shared*-Verzeichnis standardmäßig zum Schreiben von Log-Dateien (*shared/log*).

12.7.1 Eine „persistente“ Datenbank-Konfiguration

Wir demonstrieren die Verwendung des *shared*-Verzeichnisses am Beispiel einer Datenbank-Konfigurationsdatei, die aus Sicherheitsgründen nicht ins RCS eingecheckt wird.

Als Erstes wird auf dem Server ein Verzeichnis *config* unterhalb von *shared* angelegt:

```
ssh ontrack.b-simple.de -l rails
ontrack.b-simple.de:/home/rails> cd ontrack/shared
ontrack.b-simple.de:/home/rails> mkdir config
ontrack.b-simple.de:/home/rails> chmod 755 config
```

Anschließend kann die Datei *database.yml* vom lokalen Rechner ins neu angelegte *config*-Verzeichnis des Servers kopiert werden:

```
scp config/database.yml \
rails@ontrack.b-simple.de:/home/rails/ontrack/shared/config
```

OnTrack erwartet die Datenbank-Konfigurationsdatei im Verzeichnis *config* des jeweils aktuellen Deployments. Da die Datei nicht im RCS liegt, wird sie auch nicht dorthin ausgecheckt. Damit OnTrack die Datei dennoch findet, benötigt man einen Link *current/config/database.yml* auf *shared/config/database.yml*. Ein guter Anwendungsfall für einen After-Task:

```
namespace :deploy do
  desc "link in production database credentials"
  task :after_update_code do
    run <<-CMD
      rm #{release_path}/config/database.yml &&
      ln -nfs #{deploy_to}/#{shared_dir}/config/database.yml
          #{release_path}/config/database.yml
    CMD
  end
end
```

Capistrano führt den Task automatisch nach jedem Code-Update aus. Es wird sichergestellt, dass die Datei *database.yml* gefunden wird und immer auf dieselbe Datei zeigt.

Neben *log* und *config* kann auch das Verschieben des *tmp*-Verzeichnisses sinnvoll sein. Beispielsweise werden Sessions in der Standardkonfiguration ins *tmp*-Verzeichnis der Anwendung geschrieben. Ein während des Produktionsbetriebs durchgeführtes Deployment würde den *current*-Link umsetzen, was ein neues *tmp*-Verzeichnis ergäbe. Gültige Sessions wären dadurch nicht mehr verfügbar, sodass die zum Zeitpunkt des Deployments beim System angemeldeten Benutzer ausgeloggt werden.

12.8 Capistrano-Referenz

Auch wenn man Capistrano grundsätzlich verstanden und die ersten erfolgreichen Deployments durchgeführt hat, kommt es danach immer wieder vor, dass man bestimmte Konfigurationsparameter oder Cap-Tasks erneut nachschlagen muss. Aus diesem Grund wiederholen wir an dieser Stelle alle wichtigen Einstellungen und Befehle, so dass Sie bei Ihrem nächsten Capistrano-Setup direkt auf diese Seite springen und das Buch aufgeschlagen neben Ihrer Tastatur liegen lassen können.

1. Anwendung Capistrano-ready machen

```
cd application
capify .
```

2. Erforderliche Variablen in deploy.rb setzen

```
set :application, "application"
set :repository,  "svn://<svn-server>/#{application}/trunk"
set :deploy_to,   "/home/rails/#{application}"
set :use_sudo,    false
set :user,        "rails"
```

3. Erforderliche Rollen in deploy.rb setzen

```
role :web, "<ip-web-server>"
role :app, "<ip-app-server>"
role :db,  "<ip-db-server>", :primary => true
```

4. Setup des entfernten Verzeichnisses

```
cap deploy:setup
```

5. Erstes Deployment

a. Start-Task in deploy.rb definieren:

```
namespace :deploy do
  task :start do
    run "nohup #{current_path}/script/process/spawner -p 8000 -i 1"
    run "cat /home/rails/nohup.out"
    run "rm /home/rails/nohup.out"
  end
end
```

b. Deployment und Start der FastCGI-Prozesse:

```
cap deploy:cold
```

6. Datenbankmigration

```
cap deploy:migrate
```

7. Nachfolgende Deployments

```
cap deploy          # Ohne Datenbankmigration
cap deploy:migrations # Mit Datenbankmigration
```

12.9 Capistrano-Konfiguration: deploy.rb

```
# =====
# REQUIRED VARIABLES
# =====
set :application, "ontrack"
set :repository, "svn://svn.b-simple.de/#{application}/trunk"

# =====
# ROLES
# =====
role :web, "ontrack.b-simple.de"
role :app, "ontrack.b-simple.de"
role :db, "ontrack.b-simple.de", :primary => true

# =====
# OPTIONAL VARIABLES
# =====
set :deploy_to, "/home/rails/ontrack"
set :user, "rails"
set :use_sudo, false

# =====
# TASKS
# =====
desc "Spawn FastCGI-Listener"
namespace :deploy do
  task :start do
    run "nohup #{current_path}/script/process/spawner -p 8000 -i 1"
    run "cat /home/rails/nohup.out"
    run "rm /home/rails/nohup.out"
  end
end
end
```


Kapitel 13

Testgetriebene Entwicklung mit Ruby und Test::Unit

Testgetriebene Softwareentwicklung (TDD = Test Driven Development) ist eine Entwurfs- und Programmiermethode, bei der die Unit Tests vor dem eigentlichen Code entwickelt werden. Bei konsequenter Anwendung entsteht so ein dichtes Netz an Unit Tests, das Vertrauen und Sicherheit in das entwickelte System schafft. Dieses Vertrauen gibt uns den Mut, Änderungen an bestehenden Systemen vorzunehmen, ein Refactoring durchzuführen oder neue Anforderungen zu realisieren.

Dieses Kapitel führt in die Grundlagen der testgetriebenen Softwareentwicklung ein und soll Ihnen vor allem Lust aufs Testen machen.

13.1 Unit Tests – eine Definition

Ein Unit Test ist ein programmierter Test, der eine Einheit (Unit) des zu entwickelnden Softwaresystems isoliert vom Rest des Systems testet. Unit Tests werden automatisch ausgeführt und überprüfen ihre Ergebnisse selbstständig (self-testing code):

- **Automatische Ausführung:** Ein Unit Test muss auf Knopfdruck durchführbar sein. Der Test läuft dann automatisch und ohne weitere Benutzerinteraktion.
- **Selbstständige Überprüfung von Ergebnissen:** Tests müssen ihre Ergebnisse selbst überprüfen und dürfen den Entwickler nur dann informieren, wenn etwas schiefgelaufen ist.

Diese beiden Eigenschaften sind Voraussetzung dafür, dass Sie Ihre Unit Tests nach jeder Änderung am System erneut ausführen. Ist das Ausführen der Unit Tests nicht mindestens genauso einfach wie das Compilieren des Programms, werden Sie die Tests nicht nutzen.

13.2 Ein Beispiel

Da es einfacher ist, Unit Tests zu programmieren, als über deren Programmierung zu schreiben, wollen wir dieses Kapitel mit einem Beispiel beginnen. Im Folgenden wird ein Unit Test für die Klasse *Money* entwickelt. Die Klasse *Money* repräsentiert einen Geldbetrag mit einer dazugehörigen Währung. Das Beispiel ist an einen Unit Test der JUnit-Dokumentation angelehnt (siehe auch [19]).

```
class Money
  attr_reader :amount

  def initialize(amount, currency)
    @amount = amount
    @currency = currency
  end
end
```

Für die Klasse *Money* soll die Testklasse *MoneyTest* entwickelt werden. Testklassen werden in Ruby getrennt von der zu testenden Klasse definiert. In unserem Beispiel wird die Klasse *MoneyTest* in der Datei *money_test.rb* definiert.

Die Programmierung von Unit Tests ist denkbar einfach: Die Testklasse überprüft mit Hilfe von Assertions (vgl. Abschnitt 13.4), ob die zu testende Klasse bestimmte Annahmen erfüllt. Ein einfacher erster Test ist z.B. die Überprüfung, ob neue *Money*-Objekte korrekt initialisiert werden. Die zu testende Annahme lautet: Neu erzeugte Objekte sollen mit dem ihnen im Konstruktor übergebenen Betrag initialisiert werden. Der zugehörige Test kann diese Annahme prüfen, indem er zunächst ein *Money*-Objekt instanziiert. Anschließend fragt der Test das Objekt nach seinem Betrag und vergleicht diesen Wert mit dem an den Konstruktor übergebenen Wert:

```
require "test/unit"
class MoneyTest < Test::Unit::TestCase

  def test_initialize
    money = Money.new(100.0, "EUR")
    assert_equal 100.0, money.amount
  end
end
```

Ausgeführt wird der Test durch den Aufruf *ruby money_test.rb*. Haben Sie die Klasse *Money* und die zugehörige Testklasse *MoneyTest* wie beschrieben programmiert, werden Sie durch die folgende Ausgabe in Ihrem Terminal-Fenster belohnt:

```
Loaded suite money_test
Started
.
Finished in 0.0 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Um sicherzugehen, dass der Test tatsächlich leistet, was von ihm erwartet wird, verändern wir den Code der zu testenden Klasse manchmal bewusst so, dass der zu-

gehörige Test fehlschlägt. Dies ist insbesondere immer dann empfehlenswert, wenn neu entwickelter Code auf Anhieb zu korrekten Tests führt.

Im Grunde genommen beschreibt das obige Unit Test-Beispiel bereits die grundlegende Philosophie beim Schreiben von Unit Tests: Ein Test instanziiert das zu testende Objekt, ruft Methoden auf dem Objekt auf und überprüft im Anschluss an die Methodenausführung die Gültigkeit bestimmter Annahmen. Was sich hier noch relativ einfach anhört, kann in der Praxis beliebig kompliziert werden. Ein Weg zum Umgang mit dieser Komplexität ist die Anwendung der testgetriebenen Softwareentwicklung, die wir in Abschnitt 13.5 beschreiben.

13.3 Warum testen?

Früher hielten wir das Testen von Software für eine aufwändige und langweilige Tätigkeit, die am Ende des Entwicklungsprozesses steht und die man nach Möglichkeit nicht selbst ausführen möchte. Schließlich sind wir ja Softwareentwickler und keine Softwaretester. Unsere Ansicht hat sich in den letzten Jahren grundlegend geändert. Mit dem Aufkommen der agilen Softwareentwicklung, insbesondere XP, ist eine neue Art des Testens ins Bewusstsein vieler Softwareentwickler vorgedrungen: Unit-Testen.

Bei dieser Art des Testens werden die Tests nicht manuell ausgeführt, sondern programmiert und können so jederzeit automatisch erneut ausgeführt werden. Ein entscheidender Grund für Entwickler, Unit Tests zu schreiben, ist die Tatsache, dass diese Art des Testens unheimlich viel Spaß macht („Programmers love to write Tests“, siehe [19]). Denn eigentlich ist es ja kein wirkliches Testen im herkömmlichen Sinn, sondern wir programmieren die Tests, und Programmieren macht schließlich Spaß. Neben dem Spaß gibt es allerdings eine Reihe weiterer entscheidender Gründe für die Entwicklung von Unit Tests.

- **Qualität.** Ein dicht gespanntes Netz an Unit Tests stellt eine hohe Qualität des entwickelten Systems sicher. Bei konsequenter Anwendung ist jedes Stück Sourcecode getestet. Die Qualität des Systems kann jederzeit und ohne Aufwand überprüft werden, da Unit Tests automatisch und ohne Benutzerinteraktion ausgeführt werden können.
- **Refactoring.** Das Vorhandensein von Unit Tests ist eine unabdingbare Voraussetzung für Refactoring – einem weiteren Pfeiler der agilen Softwareentwicklung. Tests eröffnen uns die Möglichkeit, die interne Struktur des Systems zu verbessern, ohne das nach außen hin sichtbare Verhalten des Systems zu verändern.

Das Vorhandensein von Tests als Voraussetzung fürs Refactoring ist insbesondere für dynamische Sprachen wie Ruby von entscheidender Bedeutung, da hier ein automatisches und abgesichertes Refactoring nicht ohne weiteres möglich ist. Existieren keine Tests, verzichten wir auf Refactoring aus Angst, vorhandene Funktionalität kaputtzumachen. Das System wird so auf lange Sicht immer weniger wartbar.

- **Änderbarkeit.** Tests schaffen die Voraussetzung für änderbare und damit wartbare Software. Software lebt, wenn sie – getrieben durch neue Anforderungen – weiterentwickelt wird. Tests sorgen dafür, dass Änderungen vorhandene Funktionalität nicht zerstören. Der Ausspruch „Never touch a running system“ verliert an Bedeutung.
- **Dokumentation.** Unit Tests dokumentieren den Code und dessen Verwendung. Die Tests dienen als Beispiel, wie Klassen und Methoden des Systems zu verwenden sind. Externe Dokumentation wird nicht synchron zum Sourcecode gehalten. Unit Tests und Code sind immer synchron.
- **Feedback.** Tests geben unmittelbares Feedback auf Codeänderungen. Wir bemerken frühzeitig, dass etwas nicht mehr funktioniert, und können darauf mit entsprechenden Korrekturen reagieren. Die seit dem letzten korrekten Testlauf durchgeführten Änderungen sind überschaubar, was die Zahl der potenziellen Fehlerursachen drastisch reduziert. Stundenlange Debug-Sessions gehören der Vergangenheit an.
- **Vertrauen und Mut.** Tests schaffen Vertrauen in die vorhandene Codebasis. Wir können jeden Arbeitstag mit der Gewissheit beenden, dass unser System läuft und eine hohe Qualität aufweist. Das aus einem vorhandenen Testnetz resultierende Vertrauen gibt uns den Mut zu Veränderungen. Wenn wir etwas kaputt machen, werden wir das Problem bemerken und nicht unser Kunde.

Der Entwicklung von Unit Tests kommt im Umfeld dynamischer Sprachen eine ganz besondere Rolle zu. Ein verbreitetes Argument der Verfechter statischer Programmiersprachen wie Java ist, dass in statischen Sprachen Fehler gefunden werden, die sonst nur sehr schwer zu finden sind (siehe Blog von Martin Fowler [20]). Andererseits werden diese Typ-Fehler auch durch vorhandene Unit Tests gefunden. Da die Tests aber noch viel mehr und ganz andere Fehler finden, werden Tests sowohl in statischen als auch in dynamischen Sprachen benötigt.

Eine dynamische Sprache zwingt den Benutzer quasi zum Entwickeln von Unit Tests. Der Benutzer hat aber auch die Zeit dazu, da die Sprache selbst einfacher und damit schneller zu programmieren ist. Ein Unit-getestetes System, das in einer dynamischen Sprache entwickelt wurde, ist besser als ein in einer statischen Sprache entwickeltes System ohne Unit Tests. Testgetriebene Entwicklung entkräftet das Argument, dynamische Sprachen seien gefährlich.

Unit Tests decken allerdings nur einen Teil der benötigten Testaktivitäten ab. Sie sind kein Ersatz für Integrations- und Akzeptanztests, die auf jeden Fall Teil jeder Auslieferung der entwickelten Software sein sollten.

13.4 Test::Unit

Inspiziert durch das im Java-Umfeld weit verbreitete Testframework JUnit, wurden in den letzten Jahren ähnliche Frameworks für nahezu jede Programmiersprache entwickelt. Das Framework für Unit Tests in Ruby heißt Test::Unit und wurde von

Nathaniel Talbott entwickelt. Das Framework ist seit Ruby 1.8.1 fester Bestandteil der Distribution.

Test::Unit bietet eine API zum Programmieren individueller Unit Tests, Konstrukte zur Gruppierung und Strukturierung zusammengehöriger Tests sowie ein Framework zur automatischen Ausführung von Tests.

13.4.1 Strukturierung von Unit Tests

Test::Unit organisiert und strukturiert Unit Tests mit Hilfe verschiedener Bausteine:

1. Assertion
2. Test
3. Test Case
4. Test Suite

Assertion

Der Basisbaustein eines Unit Tests ist die Assertion, die die eigentliche Arbeit eines Tests übernimmt. Assertions formulieren Annahmen in Form erwarteter Ergebnisse. In Test::Unit sind Assertions als Methoden realisiert. Es gibt eine Reihe von Assertion-Typen, die alle mit *assert_* beginnen. Die einfachste Assertion ist *assert*, die erfüllt ist, wenn der übergebene Ausdruck *true* ist. Eine fehlschlagende Assertion wirft eine Exception. Weitere Beispiele für Assertions sind *assert_not_nil* für die Überprüfung, ob ein Objekt *nil* ist, oder *assert_equal* zum Vergleich zweier Objekte.

Test

Ein Test ist eine Methode, die eine Reihe zusammengehöriger Assertions zusammenfasst. Beispielsweise fasst der Test *test_initialize* der Test-Klasse *MoneyTest* sämtliche Assertions zusammen, die sich auf die Initialisierung von *Money*-Objekten beziehen. Als weiterer Test der *Money*-Klasse ist der Test *test_add* denkbar, der Assertions zur Überprüfung der Additionsfunktionalität von *Money* enthält.

Jede Test-Methode muss mit dem Präfix *test* starten. Dies ist zwingend erforderlich, da Test::Unit seine Tests mittels Reflection ausführt und dabei nach dieser Namenskonvention arbeitet. Die Ausführung eines Tests wird abgebrochen, sobald eine der enthaltenen Assertions fehlschlägt, d.h. eine Exception wirft.

Test Case

Ein Test Case ist ein übergeordneter Baustein zum Zusammenfassen zusammengehöriger Tests. Technisch betrachtet ist ein Test Case eine Ruby-Klasse, die von *Test::Unit::TestCase* erbt:

```
require "test/unit"
class MoneyTest < Test::Unit::TestCase
  ...
end
```

Die Funktionalität des Frameworks wird durch *require 'test/unit'* eingebunden. Ein Test Case muss mindestens einen Test enthalten. Andernfalls bricht die Ausführung des Tests mit einer Fehlermeldung ab.

Neben Test-Methoden besitzt ein Test Case zwei weitere wichtige Methoden: *setup* und *teardown*. Die Methode *setup* dient hauptsächlich der Initialisierung von Testdaten, dem so genannten Fixture des Test Cases. Ein Fixture ist eine Menge von Test-Daten, die optimalerweise von jedem Test des Test Cases benötigt werden. Das Fixture des Test Cases *MoneyTest* könnte z.B. aus der Instanzvariable *@money* bestehen:

```
class MoneyTest ...
  def setup
    @money = Money.new
  end
end
```

Da die Methode *setup* vor Ausführung jedes Tests aufgerufen wird, steht jedem Test automatisch ein frisches *Money*-Objekt zur Verfügung, ohne dass der Test das Objekt selber instanziiieren muss.

Neben der Bereitstellung des Fixtures dient die *setup*-Methode zur Initialisierung weiterer vom Test benötigter Ressourcen, wie z.B. Verbindungen zu Datenbanken. Werden derartige Ressourcen vor Ausführung eines Tests angefordert, ist es zwingend erforderlich, diese Ressourcen nach Ausführung eines Tests auch wieder freizugeben. Genau diesem Zweck dient die Methode *teardown*, die im Anschluss an die Ausführung jedes Tests aufgerufen wird.

Bei der Frage nach der Benennung scheiden sich die Geister. Der Entwickler von Test::Unit Nathaniel Talbott schlägt vor, Test Cases durch ein vorangestelltes TC zu kennzeichnen (Beispiel: TCMoney). Im Rails-Umfeld werden Test Cases durch ein nachgestelltes *Test* gekennzeichnet, d.h. der Test Case der Klasse *Money* heißt *MoneyTest* und wird in der Datei *money_test.rb* gespeichert. Wir halten uns in unseren Projekten und in unserem Buch an den letzteren Standard.

Test Suites

Test Suites fassen eine Menge zusammengehöriger Test Cases zusammen und sind ein Konstrukt zur Organisation von Test Cases. Mit Hilfe von Test Suites können mehrere Test Cases auf einmal ausgeführt werden. Technisch gesehen ist eine Test Suite eine Ruby-Klasse, die die zugehörigen Test Cases über das *require* Statement lädt:

```
require "test/unit"
require "money_test"
...
```

Die Organisation von Test Cases mit Hilfe von Test Suites ist äußerst flexibel. Projekte beginnen in der Regel mit einer einzigen Suite für alle Test Cases des Systems. Damit dabei nicht jeder neu hinzukommende Test Case explizit in die Suite eingebunden werden muss, sollte die Suite folgendermaßen programmiert werden:

```
require "test/unit"  
Dir["**/*/*_test.rb"].each { |t| require t }
```

Bei konsequenter Anwendung des Unit Test-Ansatzes wird die Zahl der Test Cases relativ schnell zunehmen, was nicht ohne Auswirkung auf die Ausführungszeit der AllTestSuite bleibt. Spätestens wenn uns diese Zeit zu lang wird, ist der Zeitpunkt gekommen, eine feinere Aufteilung der Test Cases vorzunehmen. Den Variationmöglichkeiten sind dabei keine Grenzen gesetzt. Denkbar ist z.B. eine Suite für die Domain- und eine Suite für die Datenbankzugriffsklassen eines Systems; oder in Rails-Anwendungen eine Suite für Modelle und eine für Controller.

Da Test Suites hierarchisch organisiert sind, können sie ihrerseits wieder Test Suites enthalten. Mit der Zeit entsteht so ein Baum von Suites und Tests. Die AllTestSuite bleibt natürlich vorhanden, da spätestens vor dem Einchecken von Änderungen sämtliche Unit Tests des Systems ausgeführt werden müssen.

13.4.2 Wohin mit den Tests?

Test Suites und Test Cases sollten vom Produktionscode getrennt werden und gehören deshalb in ein eigenes Verzeichnis. Dave Thomas schlägt in [3] die folgende Verzeichnisstruktur zur Organisation von Produktions- und Test-Code vor:

```
APP_ROOT  
  lib  
  test
```

Im Verzeichnis *lib* befindet sich der Produktions- und im Verzeichnis *test* der Testcode. Rails geht hier einen ähnlichen Weg, mit der Ausnahme, dass das Verzeichnis *lib* in Rails-Applikationen *app* heißt.

13.4.3 Ausführen der Tests

Die Möglichkeiten zum Ausführen von Tests sind vielfältig. Test Cases werden wie normale Ruby-Programme durch Aufruf des Interpreters, gefolgt vom Namen der Testklasse, ausgeführt:

```
ruby money_test.rb
```

Das Framework führt jede in *MoneyTest* enthaltene Testmethode mittels Reflection aus. Läuft der Test durch, wird dies durch eine Erfolgsmeldung auf der Konsole quittiert:

```
Loaded suite money_test  
Started  
..  
Finished in 0.0 seconds.  
2 tests, 4 assertions, 0 failures, 0 errors
```

Schlägt ein Test fehl, liegt dies entweder an einem Fehler im Ruby-Code oder an einer fehlschlagenden Assertion. Fehler im Ruby-Code werden bei der Testausführung

durch ein E (= Error), fehlschlagende Assertions durch ein F (= Failure) signalisiert. Zusätzlich wird eine entsprechende Fehlermeldung ausgegeben.

Test::Unit ermöglicht auch die gezielte Ausführung eines einzelnen Tests eines Test Cases:

```
ruby money_test.rb --name test_initialize
```

Der Aufruf führt nur den Test *test_initialize* aus. Diese Art der Ausführung ist dann sinnvoll, wenn Sie dediziert einen fehlschlagenden Test zum Laufen bringen wollen, die Menge der Tests des umschließenden Test Cases aber schon relativ groß ist.

13.4.4 Unabhängigkeit von Tests

Die Ausführungsumgebung von Test::Unit arbeitet mit Reflection und führt jeden Test eines Test Cases automatisch aus. Dabei erzeugt das Framework für jeden Test eines Test Cases eine neue Instanz des Test Cases. Hierdurch wird sichergestellt, dass jeder Test unabhängig von allen anderen und somit unabhängig von deren Ergebnissen ausgeführt wird. Abbildung 13.1 auf der gegenüberliegenden Seite veranschaulicht den Ablauf der Testausführung.

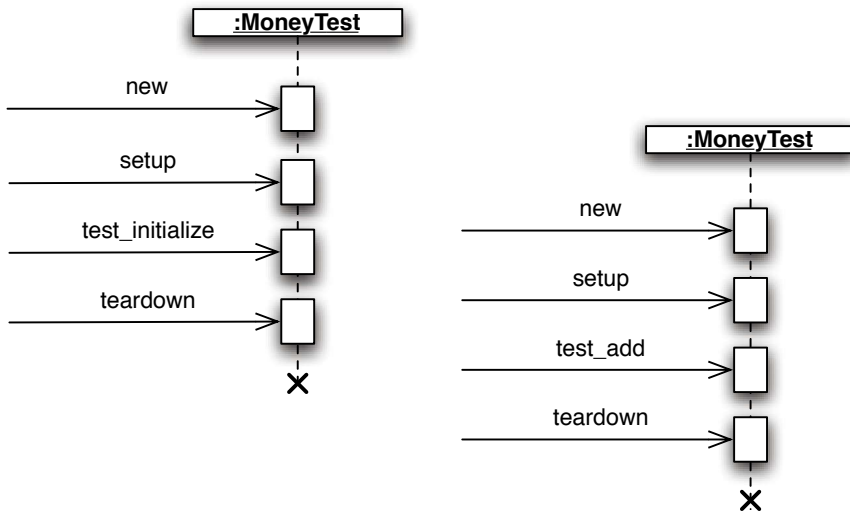


Abbildung 13.1: Isolierte Ausführung der Tests

13.5 Testgetriebene Softwareentwicklung

In den vorausgehenden Abschnitten haben wir Ihnen mit der Ruby-Bibliothek Test::Unit das nötige Handwerkszeug für die Entwicklung von Unit Tests an die Hand gegeben. In diesem Abschnitt werden wir beschreiben, wie Sie dieses Werk-

zeug für die Anwendung der testgetriebenen Softwareentwicklung (TDD = Test Driven Development) nutzen können. Doch zunächst einige Worte zum Wesen der testgetriebenen Softwareentwicklung.

Testgetriebene Softwareentwicklung bezeichnet einen Programmier- und Designstil, der die Entwicklung von Unit Tests an den Anfang jeglicher Programmieraktivitäten stellt. Konkret bedeutet dies, dass ein Test für die neue Funktionalität vorhanden sein muss, bevor überhaupt eine Zeile Code der zu entwickelnden Klasse geschrieben wird.

Testgetriebene Softwareentwicklung ist ein iterativer Prozess, der im Kern aus der Wiederholung folgender Arbeitsschritte besteht:

1. Schreibe einen fehlschlagenden Test.
2. Schreibe gerade so viel Code, dass der neue Test und alle bereits vorhandenen fehlerfrei ausgeführt werden können.
3. Entferne Code-Duplizierung und andere Code Smells.¹

Test. Der erste Schritt dieses Kreislaufs ist der eigentlich neue und vielleicht etwas verwirrende Arbeitsschritt. Sie fragen sich vielleicht, wie Sie einen Test für etwas entwickeln sollen, das es noch gar nicht gibt. Ganz einfach: Sie schreiben den Test genau deshalb, weil Sie noch nicht wissen, wie die Implementierung der neuen Klasse bzw. Methode aussehen wird.

Die Entwicklung des Tests vor dem Code zwingt Sie dazu, einen äußeren Standpunkt einzunehmen und darüber nachzudenken, wie die zu entwickelnde Klasse möglichst testbar programmiert werden kann. Seitdem wir testgetrieben entwickeln, verbringen wir viel mehr Zeit damit, über den zu entwickelnden Code nachzudenken, bevor wir den Code wirklich entwickeln. Und genau das ist der Punkt. Erst wenn wir das zu lösende Problem ausreichend verstanden haben, sind wir in der Lage, einen Test dafür zu formulieren.

Code. Weil die neue Funktionalität noch nicht vorhanden ist, schlägt der Test bei seiner ersten Ausführung fehl. Schritt 2 beschreibt einen zentralen Grundsatz der testgetriebenen Softwareentwicklung: Es wird nur neuer Code entwickelt, wenn es mindestens einen fehlschlagenden Test gibt.² Fehlschlagende Tests sind der Motor der testgetriebenen Softwareentwicklung. Sie treiben das Design der Anwendung und zwingen den Entwickler, den Code so zu verändern oder zu erweitern, dass der Test fehlerfrei durchläuft. Um sicherzugehen, dass der neue Code keine vorhandene Funktionalität zerstört, muss der Entwickler dafür sorgen, dass sämtliche schon vorhandenen Tests weiterhin fehlerfrei durchlaufen.

Refactoring. Schlussendlich erfolgt der letzte Schritt des TDD-Kreislaufs: das Aufräumen. Während in Schritt 2 der Fokus ausschließlich auf der fehlerfreien Ausführbarkeit der Tests liegt, geht es in diesem Schritt darum, ein wartbares System zu entwickeln.

¹ Übel riechender Code im Sinne von notwendiger Bereinigung durch Refactoring.

Siehe <http://xp.c2.com/CodeSmell.html>

² Wie wir später noch sehen werden, ist Refactoring eine Ausnahme, die das Abweichen von dieser Regel rechtfertigt.

Der Test-Code-Refactoring-Kreislauf ist bewusst sehr kurz und wird ständig wiederholt. Dies hat den Vorteil, dass immer nur sehr wenig und damit schnell und einfach nachvollziehbare Änderungen vorgenommen werden. Kommt es dabei zu Fehlern oder Seiteneffekten, sind diese genauso schnell eingrenzbar und können unmittelbar beseitigt werden.

Der TDD-Kreislauf treibt den Entwickler von Entwurfsentscheidung zu Entwurfsentscheidung. Testgetriebene Softwareentwicklung ist damit viel mehr als eine reine Qualitätssicherungsmaßnahme, sie ist eine Entwurfsmethode, die auch als evolutionäres Design bezeichnet wird. Auf die Durchführung einer Designphase vorab (BDUF = Big Design Up-Front) wird zugunsten einer kontinuierlichen Verbesserung des Designs verzichtet (siehe [2]).

13.5.1 TODO-Listen

Der TDD-Kreislauf besteht aus vielen kleinen und restriktiven Schritten. Z.B. dürfen wir immer nur so viel Code schreiben, dass alle Tests fehlerfrei laufen. Oft bedeutet dies, dass wir aus bestimmten Methoden einfach nur einen konstanten Wert zurückliefern, da ein neuer Test diesen Wert mit einem angenommenen Wert vergleicht. Restriktiv meint hier also, dass wir uns darüber im Klaren sind, dass die aktuelle Lösung nicht ausreicht, wir aber keine allgemeingültige Lösung programmieren dürfen.

Weil wir Dinge, die uns auffallen, nicht sofort in die Tat umsetzen dürfen, müssen wir uns diese Punkte notieren. Zu diesem Zweck schlägt Kent Beck in [1] die Verwendung von TODO-Listen vor. Immer dann, wenn Ihnen während der Programmierung etwas auffällt, fügen Sie Ihrer TODO-Liste einen weiteren Eintrag hinzu. Dies können z.B. notwendige neue Tests, Ideen für Refactorings oder auch neue Anforderungen sein. Hier einige Beispiele:

- Erweitere `test.add` um eine Assertion, die einen anderen Wert als 2 addiert.
- Deklariere die Methode `round` als `private`.
- Erweitere die Klasse `Money` um eine Methode `sub()`.

Die TODO-Liste sorgt dafür, dass Dinge nicht in Vergessenheit geraten. Am Ende eines TDD-Zyklus werfen Sie einen Blick auf die Liste und nehmen sich den Punkt mit der höchsten Priorität vor.

13.5.2 Beispiel

Ein Beispiel soll den TDD-Ansatz verdeutlichen. Die `Money`-Klasse soll testgetrieben um eine `add`-Methode erweitert werden. Die Methode erhält als Parameter ein `Money`-Objekt und soll als Ergebnis die Summe der Beträge beider Objekte liefern.

Da wir keinen neuen Code schreiben, solange wir nicht mindestens einen fehlschlagenden Test haben, beginnen wir mit der Entwicklung des Tests `test.add`:

```
class MoneyTest ...
```

```

def test_add
  sum = @money.add Money.new(2, "EUR")
  assert_equal Money.new(102, "EUR"), sum
end
end

```

Die Ausführung des Tests liefert uns ein *E*, d.h. einen Ruby-Fehler, da die Methode *add* bisher noch nicht definiert wurde:

```

Loaded suite money_test
Started
E.
Finished in 0.0 seconds.

1) Error:
test_add(MoneyTest):
NoMethodError: undefined method `add' for
#<Money:0x2b1eb50 @currency="EUR", @amount=100.0>
money_test.rb:15:in `test_add'

2 tests, 1 assertions, 0 failures, 1 errors

```

Kent Beck schlägt in [1] vor, nur genau so viel Code zu schreiben, wie nötig ist, um den neuen Test zum Laufen zu bringen³. Für unser Beispiel reicht es also aus, die Methode *add* zunächst so zu implementieren, dass sie immer eine konstante *Money*-Instanz mit dem vom Test erwarteten Wert zurückliefert:

```

class Money ...
  def add(money)
    Money.new(102, currency)
  end
end

```

Nach dem Hinzufügen dieser Methode kann der Test fehlerfrei ausgeführt werden. Eine brauchbare Implementierung der *Money*-Klasse sollte jedoch auch in der Lage sein, andere Werte als 100 und 2 addieren zu können. Eine weitere Assertion zwingt uns zu den dafür notwendigen Änderungen:

```

class MoneyTest ...
  def test_add
    sum = @money.add Money.new(2, "EUR")
    assert_equal Money.new(102, "EUR"), sum
    sum = @money.add Money.new(23, 'EUR')
    assert_equal Money.new(123, 'EUR'), sum
  end
end

```

Der Test zwingt uns, die konstante Implementierung der *add*-Methode aufzugeben und durch eine „richtige“ Implementierung zu ersetzen.

³ Johannes Link liefert hierfür eine Begründung auf S. 39 seines Buches *Softwaretests mit JUnit* (vgl. [2]).


```
class Money ...  
  def add money  
    Money.new(102, currency)  
    Money.new(amount + money.amount, currency)  
  end  
end
```

Der Test läuft, und die testgetriebene Entwicklung der *add*-Methode ist damit abgeschlossen.⁴

13.6 Retrospektive

Wie Sie gesehen haben, ist es gar nicht so schwer, einen Test für etwas zu entwickeln, das es noch gar nicht gibt. Üben Sie sich in der testgetriebenen Entwicklung. Die konsequente Anwendung des Test-Code-Refactoring-Kreislaufs wird sich innerhalb kürzester Zeit automatisieren und ist dann völlig selbstverständlich. Sie werden durch ein bisher nicht gekanntes Vertrauen in Ihren Quellcode belohnt, schließlich ist jedes Stück Funktionalität getestet.

⁴ In einer richtigen Implementierung muss die Methode *add* die Währung überprüfen und ggf. mit einem Fehler oder einer vorherigen Umrechnung reagieren.

Kapitel 14

Testgetriebene Entwicklung mit Ruby on Rails

Neben Einfachheit und Produktivität liegt ein weiterer Schwerpunkt von Rails auf der Entwicklung von testbaren Web-Anwendungen. Rails zwingt den Entwickler zwar nicht zum Entwickeln von Unit Tests, weist ihn aber sanft darauf hin, indem für Modell- und Controllerklassen automatisch zugehörige Testklassen generiert werden. Die generierten Testklassen sind zunächst nicht viel mehr als ein Skelett, das zum Leben erweckt werden muss, indem es mit konkreten Tests gefüllt wird.

Wir empfehlen Ihnen, jede Rails-Applikation vollständig und lückenlos durch Unit Tests abzusichern. Was zunächst aufwändig klingt, zahlt sich mittel- und langfristig auf jeden Fall aus. Und da es einfacher ist, Tests vor der eigentlichen Funktionalität zu entwickeln, sollte die Entwicklung testgetrieben erfolgen. Dieses Kapitel zeigt, wie das machbar ist.

Das Kapitel beginnt mit einem Fallbeispiel. Wir zeigen am Beispiel eines Haushaltsbuchs, wie die einzelnen Komponenten einer Rails-Applikation testgetrieben entwickelt werden. Geänderte Anforderungen zwingen uns später, das System umzubauen, um es für die Änderungen und Erweiterungen zu öffnen. Wir zeigen, wie sich diese Änderungen dank der zuvor entwickelten Tests sicher durchführen lassen.

Der zweite Teil dieses Kapitels beschreibt die im Rahmen des Fallbeispiels verwendeten Testkonzepte und die von Rails zur Verfügung gestellte Test-API im Detail. Testkonzepte für die Implementierung von Tests für typische Probleme runden das Kapitel ab.

14.1 Generierte Testklassen

Die Verzeichnisstruktur für die von Rails generierten Testklassen sieht wie folgt aus:

```
APP_ROOT/test
  fixtures
```

```
functional
integration
mocks
unit
```

Tests für Modellklassen landen im Verzeichnis *unit* und heißen wie die Modellklasse, gefolgt von *_test.rb*. Tests für Controllerklassen und Views werden ins Verzeichnis *functional* generiert und genügen demselben Namensschema. Zusätzlich generiert Rails für jeden Modelltest eine Fixture-Datei im Verzeichnis *fixtures*. Rails-Fixtures sind Textdateien mit Testdaten, die vor jedem Testlauf automatisch in die Datenbank geschrieben werden. Eine ausführliche Beschreibung von Fixtures finden Sie in Abschnitt 14.5.3.

14.2 Testdatenbank

Rails-Modellklassen sind eng an ihre jeweilige Datenbanktabelle gekoppelt. Dies führt dazu, dass ein Großteil der Unit Tests für Modellklassen auf eine laufende Datenbank mit den für die Tests erforderlichen Testdaten angewiesen ist. Um dieser Forderung gerecht zu werden, stellt Rails jeder Modellklasse eine Menge von Testdaten in Form eines Fixtures zur Verfügung. Vor Ausführung jedes Tests einer Testklasse werden alle Datensätze des Fixtures in die zugehörigen Tabellen geladen. Damit die Tests bei jeder Ausführung unter denselben Bedingungen ablaufen, werden vor dem Einfügen alle vorhandenen Datensätze aus den betroffenen Tabellen gelöscht.

Würden die Unit Tests gegen dieselbe Datenbankinstanz wie die Entwicklungsversion des Systems ausgeführt, wären sämtliche interaktiv erfassten Datensätze nach Ausführung des Tests nicht mehr vorhanden und durch Fixture-Daten ersetzt worden. Dieses Verhalten ist natürlich nicht erwünscht und wird von Rails durch die Bereitstellung einer dedizierten Testdatenbank vermieden. Man konfiguriert die Testdatenbank genau wie die Entwicklungsdatenbank in der Konfigurationsdatei *APP_ROOT/config/database.yml*:

```
test:
  adapter: mysql
  database: household_test
  host: localhost
  username: rails
  password: rails
...
```

Nachdem die Datenbank konfiguriert wurde, muss sie natürlich auch erstellt werden:

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

mysql> create database household_test;
```

Die Unit Tests einer Rails-Anwendung werden üblicherweise durch Aufruf von *rake* (siehe Abschnitt 14.3) ausgeführt. Die Rake-Tasks zum Testen einer Rails-Anwendung sind so gebaut, dass vor Ausführung der Tests die Tabellen der Testdatenbank auf Basis der Developmentdatenbank neu erzeugt werden. Um Ihre Tests im Folgenden also reibungslos auszuführen, benötigen Sie neben der erstellten Testdatenbank eine Development-Datenbank:

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

```
mysql> create database household_development;
```

Die Developmentdatenbank wird analog zur Testdatenbank in der Datei *database.yml* konfiguriert.

14.3 Testausführung mit Rake

Unit Tests sollten mit *rake* ausgeführt werden. In der Standard-Rakedatei einer Rails-Anwendung *rakefile* stehen dafür vier Tasks zur Verfügung: *test:functionals* für die Ausführung der funktionalen Tests, *test:units* für die Ausführung der Unit Tests *test:integration* für die Ausführung der Integrationstests und *test:recent* für die Ausführung von Tests, für unmittelbar zurückliegende Änderungen. Interessant ist die Arbeitsweise des Tasks *test:recent*: Der Task durchsucht den Quellcode des Projekts nach Änderungen, die jünger als 10 Minuten sind, und führt für alle gefundenen Dateien den zugehörigen Test aus, sofern er vorhanden ist. Die Ausführung der Tests erfolgt auf der Kommandozeile durch den Aufruf von *rake*, gefolgt vom jeweiligen Task, z.B.:

```
rake test:functionals
```

Ein wesentlicher Vorteil bei der Verwendung von Rake ist die Tatsache, dass alle Test-Tasks von dem Task *clone_structure_to_test* abhängig sind, d.h. *clone_structure_to_test* wird vor dem eigentlichen Task ausgeführt. Der Task erzeugt eine Kopie der Entwicklungsdatenbank, indem alle darin enthaltenen Tabellen in die Testdatenbank kopiert werden. Änderungen werden also nur noch auf der Entwicklungsdatenbank ausgeführt, und die Synchronisation beider Datenbanken bleibt Rake überlassen.

14.4 Eine Programmierepisode

Wir wollen Ihnen die testgetriebene Rails-Entwicklung am Beispiel eines einfachen Haushaltsprogramms demonstrieren. Das Programm soll dem Benutzer das Verbuchen von ein- und ausgehenden Zahlungen auf verschiedene Konten ermöglichen. Diese Entwicklung wird verdeutlichen, wie uns der testgetriebene Ansatz von Entwurfsentscheidung zu Entwurfsentscheidung leitet und so das evolutionäre Design der Anwendung steuert.

Wenn Sie die Beispiele dieses Kapitels am Rechner nachvollziehen wollen, ist jetzt der richtige Zeitpunkt für die Erstellung des zugehörigen Rails-Projekts:

```
$ rails household
$ cd household
```

Die Konfiguration der Test- und Development-Datenbank erfolgt wie in Abschnitt 14.2 beschrieben in der Datei *household/config/database.yml*.

14.4.1 Entwicklung einer Modellklasse

Wir beginnen mit der testgetriebenen Programmierung der Modellklasse *Account*. Die Klasse modelliert ein Konto zur Verbuchung von Ausgaben. Sie wird durch Ausführung des Rails-Skripts *generate* erzeugt:

```
$ ruby script/generate model account
```

Neben der eigentlichen Klasse *Account* generiert das Skript die Testklasse *AccountTest* in *test/unit*. Wie bereits erwähnt, handelt es sich dabei nur um einen Rahmen, der mit Leben gefüllt werden muss. Aber auch ohne zusätzliche Bearbeitung eignet sich der Test Case schon jetzt als Indikator dafür, dass etwas nicht in Ordnung ist. Führen Sie den Test aus, und Sie werden sehen, was fehlt:

```
$ rake test:units
```

Bisher haben wir nur das Modell, nicht jedoch die zugehörige Datenbanktabelle angelegt. Der Test meldet dies mit einer eindeutigen Fehlermeldung:

```
$ rake test:units

1) Error:
test_truth(AccountTest):
ActiveRecord::StatementInvalid: Mysql::Error:
#42S02Table 'household_test.accounts' doesn't exist
...
```

Getreu dem TDD-Motto „Kein neuer Code ohne fehlschlagenden Test“ gilt es jetzt also, aktiv zu werden. Wir erstellen die zu *Account* gehörende Datenbanktabelle *accounts*. Falls noch nicht geschehen, ist nun der richtige Zeitpunkt, die Datenbank *household_test* anzulegen und in der Datei *database.yml* als Testdatenbank zu konfigurieren (siehe auch Abschnitt 14.2). Das Migrationsskript zum Anlegen der Tabelle *accounts* sieht dann wie folgt aus:

```
class CreateAccounts < ActiveRecord::Migration
  def self.up
    create_table :accounts do |t|
      end
  end

  def self.down
    drop_table :accounts
  end
end
```

Migrieren Sie die Datenbank (*rake db:migrate*), und führen Sie den Test erneut aus:

```
$ rake test:units
...
Finished in 0.030626 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Prima, der Test läuft durch – Sie haben die Klasse *Account* testgetrieben ins Leben programmiert.

Neue Attribute

Wie geht es jetzt weiter? Bisher besteht die Klasse *Account* ausschließlich aus einem Attribut *id* und ergibt in dieser Form noch nicht allzu viel Sinn. Die Klasse *Account* repräsentiert ein Konto und benötigt mindestens eine weitere Instanzvariable *balance* zum Speichern des aktuellen Kontostands. Testen lässt sich die korrekte Initialisierung des Kontostands durch einen Test, der ein *Account*-Objekt aus der Datenbank lädt und dessen Kontostand mit einem erwarteten Wert vergleicht:

```
class AccountTest < ActiveSupport::TestCase
  def test_balance
    assert_equal 3000.0, accounts(:first).balance
  end
end
```

Der Test schlägt – wie nicht anders zu erwarten – mit einem *Error* fehl, da die Methode *balance* bisher nicht implementiert wurde.

```
$ rake test:units
E
Finished in 0.030337 seconds.

1) Error:
test_balance(AccountTest):
NoMethodError: undefined method `balance'
```

In Rails werden Attribute von Modellklassen in der zugehörigen Datenbanktabelle definiert. Für die *Account*-Klasse ist es also erforderlich, die Tabelle *accounts* um das Feld *balance* zu erweitern:

```
class AddBalanceToAccounts < ActiveRecord::Migration
  def self.up
    add_column :accounts, :balance, :float
  end

  def self.down
    remove_column :accounts, :balance
  end
end
```

Diese Erweiterung sorgt für die Behebung des Ruby-Fehlers (fehlende Methode), reicht für einen erfolgreichen Testlauf aber noch nicht aus.

```
$ rake test:units
F
Finished in 0.037286 seconds.
```

```
1) Failure:
test_balance(AccountTest) [./test/unit/account_test.rb:7]:
<3000.0> expected but was
<nil>.
```

Uns steht auf der *Account*-Klasse jetzt zwar die Methode *balance* zur Verfügung, doch haben wir den Kontostand bisher an keiner Stelle initialisiert. In Rails erfolgt die Bereitstellung und Initialisierung von Testdaten über Fixtures. Was das genau ist und wie Sie damit arbeiten, werden wir später genauer erklären. An dieser Stelle sei nur so viel erwähnt, dass Rails für jede Testklasse eine Fixture-Datei im Verzeichnis *test/fixtures* erzeugt. Die Datei hat den Namen der Modellklasse (in der Pluralform), gefolgt vom Suffix *.yml*. Initial enthält die Datei ausschließlich Kommentare:

```
# Read about fixtures at http://ar.rubyonrails.org/classes/
# Fixtures.html

# one:
#   column: value
#
# two:
#   column: value
```

Unser Test verwendet das *Account*-Objekt *:first*. Soll dieses Konto mit einem Startbetrag von 3000 Euro initialisiert werden, gilt es, den entsprechenden Eintrag im Fixture zu erstellen:

```
first:
  balance: 3000.0
```

Zusätzlich muss das Fixture im Testcase geladen werden. Dies geschieht über den Aufruf der Klassenmethode *fixtures*, gefolgt vom symbolischen Namen der zu ladenden Fixturedatei:

```
require File.dirname(__FILE__) + '/../test_helper'

class AccountTest < ActiveSupport::TestCase
  fixtures :accounts
  ...
end
```

Die erneute Ausführung des Tests endet jetzt mit 0 Fehlern und 0 fehlschlagenden Assertions. Wir haben die Klasse *Account* testgetrieben um das Attribut *balance* erweitert. Dabei haben wir die einzelnen Schritte bewusst sehr detailliert beschrieben. Das mag Ihnen vielleicht ein wenig umständlich vorkommen. Wenn Sie sich aber erst einmal mit dem testgetriebenen Entwicklungsstil vertraut gemacht haben, werden Sie die einzelnen Schritte sehr schnell und nahezu automatisch durchführen

und dabei auf einzelne Zwischenschritte verzichten. Für das Hinzufügen eines neuen Attributs sollten Sie zwar immer einen neuen Test schreiben, Sie können aber das Hinzufügen der Testdaten und das Ändern der Datenbank in einem Schritt zusammenfassen.

Neue Geschäftslogik

Als Nächstes soll die Klasse *Account* um eine Methode zum Verbuchen von Ausgaben erweitert werden. Eine Ausgabe wird durch die Klasse *Disbursement* modelliert. Die Klasse besteht in ihrer Initialversion aus einer ID und einem zu verbuchenden Betrag. Eine verbuchte Ausgabe wird dem zugehörigen Buchungskonto zugeordnet. Die *Disbursement*-Klasse benötigt also ein weiteres Attribut *account_id* sowie eine *belongs_to: account*-Relation auf Modellebene:

```
$ ruby script/generate model disbursement amount:float \
    account_id:integer

class CreateDisbursements < ActiveRecord::Migration
  def self.up
    create_table :disbursements do |t|
      t.float :amount
      t.integer :account_id

      t.timestamps
    end
  end

  def self.down
    drop_table :disbursements
  end
end

class Disbursement < ActiveRecord::Base
  belongs_to :account
end
```

Vergessen Sie nicht, die Datenbank zu migrieren:

```
$ rake db:migrate
```

Das Verbuchen von Ausgaben wird von der Methode *Account.book* implementiert, die als Parameter ein *Disbursement*-Objekt erhält. Die Methode weist dem *Disbursement* eine Referenz auf das aktuelle Konto zu und subtrahiert den Betrag des *Disbursements* vom aktuellen Kontostand. Der zugehörige Test sieht folgendermaßen aus:

```
class AccountTest < ActiveSupport::TestCase
  fixtures :accounts

  def test_book
    disbursement = Disbursement.new
```



```

    disbursement.amount = 100.0
    accounts(:first).book(disbursement)
    assert_equal 2900.0, accounts(:first).balance
    assert_equal accounts(:first), disbursement.account
  end
  ...
end

```

Der Test *test.book* erzeugt ein neues *Disbursement* und initialisiert dessen Betrag mit 100 Euro. Anschließend wird die Methode *book* des *Account*-Objekts aus dem Fixture aufgerufen, die als Parameter das *Disbursement*-Objekt erhält. Der Test überprüft zwei Annahmen:

1. Wurde der Kontostand korrekt um den Betrag des Disbursements reduziert?
2. Enthält das Disbursement eine Referenz auf das Buchungskonto?

Die Ausführung des Tests schlägt fehl, da die Methode *book* noch gar nicht existiert. Holen wir das nach:

```

class Account < ActiveRecord::Base
  def book(disbursement)
    self.balance -= disbursement.amount
    disbursement.account = self
  end
end

```

Die Implementierung der eigentlichen Funktionalität ist fast einfacher als die des zugehörigen Tests: Der Kontostand wird um den Wert des übergebenen Disbursements reduziert, und dem Disbursement wird eine Referenz auf das Zielkonto zugewiesen, d.h. auf das aktuelle Objekt *self*.

14.4.2 Entwicklung des Controllers

Bisher haben wir uns ausschließlich auf die Modellklassen, d.h. auf das Domain-Modell unseres Haushaltsbuchs, konzentriert. Über kurz oder lang wird jedoch beim Benutzer der Wunsch aufkommen, neue Daten zu erfassen und einen Überblick über die bereits existierenden Daten zu bekommen. Lassen Sie uns mit dem Anzeigen einer Liste aller Konten beginnen. Dazu benötigen wir einen neuen Controller, der die benötigte Action implementiert:

```

ruby script/generate controller account

```

Neben der eigentlichen *AccountController*-Klasse erzeugt das Skript im Verzeichnis *test/functional* die Testklasse *AccountControllerTest*. Die Entwicklung der *list*-Action erfolgt wie immer testgetrieben und beginnt demzufolge mit einem neuen Test *test.should_get_index*.

Controller-Tests simulieren Web-Clients, indem sie HTTP-Requests an den Controller senden. Rails stellt hierfür fünf verschiedene Request-Typen zur Verfügung, die wir später genauer beschreiben werden. Für das aktuelle Beispiel ist das Verständnis

des *get*-Requests ausreichend. Requests werden durch den Aufruf einer gleichnamigen Methode, gefolgt von der angeforderten Action (als Symbol), ausgeführt. Eine initiale Version des Tests *test_should_get_index* könnte also folgendermaßen aussehen:

```
require File.dirname(__FILE__) + '/../test_helper'

class AccountControllerTest < ActionController::TestCase
  def test_should_get_index
    get :index
  end
end
```

Der Test sendet einen HTTP-Request vom Typ *index* an den *AccountController*. Beachten Sie, dass hierzu der Webserver nicht laufen muss. Die Ausführung des Tests schlägt mit einem Ruby-Fehler fehl:

```
$ rake test:functionals
E
Finished in 0.105354 seconds.

1) Error:
test_should_get_index(AccountControllerTest):
ActionController::UnknownAction: No action responded to index
```

Logisch, bisher haben wir ja auch noch keine *index*-Methode implementiert. Holen wir dies an dieser Stelle nach:

```
class AccountController < ApplicationController
  def index
    render :text => "Index Action"
  end
end
```

Nach dieser Erweiterung des *AccountControllers* läuft der Test fehlerfrei. Das Senden von *get*-Requests ist nur einer von wenigen Schritten beim Testen von Controllerklassen. In Abschnitt 14.6 werden die verschiedenen Requests zum Testen von Controllern genauer beschrieben.

14.4.3 Programmierung des Views

Das Rendern eines Texts aus einer Action heraus ist eine einfache, aber häufig wenig nützliche Art, eine Action zum Laufen zu bekommen. In den seltensten Fällen wird man es dabei belassen, sondern stattdessen die schnell herunter programmierten Actions mit zugehörigen Views ausstatten.

Als Beispiel für die testgetriebene Programmierung eines Views soll der *index*-Text durch einen manuell programmierten View ersetzt werden. Dafür löschen wir als Erstes den *render*-Aufruf aus der *index*-Action:

```
class AccountController < ApplicationController
  def index
```

```

    end
  end

```

Die Änderung zerbricht das System, da der am Ende einer Action zu rendernde Default-View *index.html.erb* nicht gefunden wird:

```

$ rake test:functionals
E
Finished in 0.078581 seconds.

1) Error:
test_should_get_index(AccountControllerTest):
ActionController::MissingTemplate:
Missing template account/index.html.erb in view path
/Users/ralf/tmp/household/app/views

```

Im nächsten Schritt wird der View *index.html.erb* im Verzeichnis *APP_ROOT/app/views/account* erstellt. Er kann zunächst leer bleiben, da unser primäres Ziel die Reaktivierung des Tests ist. Die erneute Ausführung des Tests bestätigt uns, dass das Hinzufügen des leeren Views dafür völlig ausreichend ist.

Zum Anzeigen von Daten benötigen Views Instanzvariablen, die von ihrem jeweiligen Controller deklariert und initialisiert werden müssen. Der folgende Test stellt sicher, dass dem *index*-View eine Liste von Accounts in der Instanzvariable *accounts* zur Verfügung steht:

```

class AccountControllerTest < ActionController::TestCase
  def test_should_get_index
    get :index
    assert_not_nil assigns(:accounts)
  end
end

```

Der Test verwendet die Getter-Methode *assigns*, die eine Hash mit sämtlichen Instanzvariablen des Controllers liefert. Mit Hilfe der Assertion *assert_not_nil* überprüft der Test, ob die Hash die Variable *accounts* enthält. Ausführliche Informationen zum Thema Collections finden Sie in Abschnitt 14.6.4.

Da die Action *index* bisher nur implementiert wurde, ohne irgendetwas zu tun, schlägt der Test wieder fehl. Erweitern Sie die Methode *index* um die Initialisierung von *accounts*:

```

class AccountController < ApplicationController
  def index
    @accounts = Account.find(:all)
  end
end

```

Der Test läuft wieder. Als Nächstes steht die Ausgestaltung des Views auf dem Programm. Dazu muss der View über die Liste der Accounts iterieren und deren Inhalte mittels entsprechender HTML-Elemente darstellen. Auch an diesem Punkt lässt Rails Sie nicht alleine. Mit Hilfe der Assertion *assert_tag* können Sie Struktur und

Inhalt von RHTML-Views testgetrieben entwickeln, ohne dabei ständig den Browser benutzen zu müssen. Wollen Sie z.B. sicherstellen, dass die Konten innerhalb einer Tabelle dargestellt werden, sichert folgende Assertion zunächst die Existenz der Tabelle ab:

```
class AccountControllerTest < ActionController::TestCase
  def test_should_get_index
    get :index
    assert_not_nil assigns(:accounts)
    assert_tag :tag => "table"
  end
end
```

Die Möglichkeiten der *assert_tag*-Assertion sind vielfältig und werden in Abschnitt 14.6.5 detailliert beschrieben.

14.4.4 Geänderte Anforderungen

Das aktuelle Design unseres Haushaltsbuchs ermöglicht die Verbuchung von Ausgaben auf ein einziges Konto. Genau wie im wirklichen Software-Leben wird auch für unsere Beispielanwendung über kurz oder lang die Anforderung entstehen, eine Ausgabe auf mehr als ein Konto zu verbuchen. Kaufen Sie beispielsweise in einem Schreibwarenladen Büroartikel, von denen Sie die Hälfte für Ihre private Korrespondenz und die andere Hälfte für Ihre Büroarbeit benötigen, dann freut sich Ihr Steuerberater, wenn Sie diese Ausgabe auf zwei unterschiedliche Konten verbuchen: ein Konto für Privatausgaben und ein Konto für Arbeitsmittel. Bevor diese Anforderung realisiert werden kann, muss die Anwendung zunächst angepasst (*refactored*) werden.

Refactoring

Im Domain-Modell des Haushaltsbuchs ist eine Ausgabe (Disbursement) genau einem Konto zugeordnet. Bevor wir die neue Anforderung umsetzen können, müssen wir das System dahingehend refaktorisieren, dass das Domain-Modell die Buchung einer Ausgabeposition auf mehr als ein Konto zulässt (siehe Abbildung 14.1).

Ziel des Refactorings ist die Veränderung der internen Struktur der Anwendung, ohne dabei ihr nach außen sichtbares Verhalten zu verändern. Wir lassen den Buchungstest *test_book* zunächst so, wie er ist, d.h. es wird weiterhin nur die Buchung auf ein einziges Konto getestet. Dennoch verändern wir das Datenmodell hinsichtlich der neuen Anforderung und prüfen, ob der Test weiterhin läuft. Ihnen wird vielleicht aufgefallen sein, dass wir hier eine grundlegende Regel der testgetriebenen Entwicklung verletzen. Wir ändern Code, ohne einen fehlschlagenden Test zu haben. Im Fall eines Refactorings ist dies jedoch zulässig.

Im neuen Datenmodell werden Buchungen von der Modellklasse *Booking* modelliert. Für die Erzeugung von Booking-Objekten ist die Methode *book* der Klasse *Account* zuständig. Wir notieren uns diesen Punkt auf unserer TODO-Liste:

- Prüfen, ob *Account.book* ein neues Booking-Objekt anlegt.

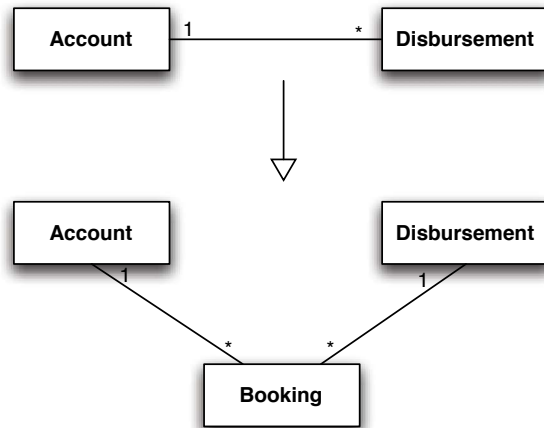


Abbildung 14.1: Refactoring des Haushaltsbuchs

Die Klasse *Booking* besitzt zwei Referenzen: eine auf *Account* und eine auf *Disbursement* (vgl. auch Abbildung 14.1). Im Datenmodell benötigen wir dafür eine Tabelle *bookings* mit zwei Fremdschlüsseln auf die referenzierten Objekte. In der Klasse *Booking* werden die Relationen durch zwei *belongs_to*-Relationen modelliert:

```

$ ruby script/generate model booking account_id:integer
  disbursement_id:integer

class CreateBookings < ActiveRecord::Migration
  def self.up
    create_table :bookings do |t|
      t.integer :account_id
      t.integer :disbursement_id

      t.timestamps
    end
  end

  def self.down
    drop_table :bookings
  end
end

class Booking < ActiveRecord::Base
  belongs_to :account
  belongs_to :disbursement
end

```

Wenn wir den *AccountTest* jetzt ausführen, ist nach wie vor alles in Ordnung, da wir bisher noch nichts an den bestehenden Modellrelationen geändert haben. Da die Beziehung zwischen *Account* und *Disbursement* jetzt jedoch von der Klasse *Booking*

modelliert wird, können wir die Referenz von *Disbursement* auf *Account* aus der Datenbank und aus der *Disbursement*- Klasse löschen:

```
class Disbursement < ActiveRecord::Base
  belongs_to :account
end
```

Damit klar wird, warum der Test nach dieser Änderung fehlschlägt, rufen wir uns zunächst noch einmal die Implementierung der *book*-Methode in Erinnerung:

```
class Account < ActiveRecord::Base
  def book(disbursement)
    self.balance -= disbursement.amount
    disbursement.account = self
  end
end
```

Jetzt ist klar: Die erneute Ausführung des Tests schlägt fehl, weil aus der Klasse *Disbursement* die automatisch von der *belongs_to*-Relation erzeugte Setter-Methode *account=* gelöscht wurde. Die Methode *book* referenziert die Methode *account=* jedoch weiterhin, was der Grund für den fehlschlagenden Test ist. Die zugehörige Fehlermeldung lautet:

```
$ rake test:units
E...
Finished in 0.081945 seconds.

1) Error:
test_book(AccountTest):
NoMethodError: undefined method 'account=' for
#<Disbursement:0x12a7244>
```

Grund genug für uns, aktiv zu werden. Höchste Priorität besitzt dabei die Reaktivierung aller fehlschlagenden Tests. Dazu löschen wir zunächst die Überprüfung der *Account*-Relation in *Disbursement* im *AccountTest*, da diese Relation im geänderten Datenmodell keine Gültigkeit mehr besitzt:

```
def test_book
  disbursement = Disbursement.new
  disbursement.amount = 100.0
  @account.book disbursement
  assert_equal 2900.0, @account.balance
  assert_equal accounts(:first), disbursement.account
end
```

Aus demselben Grund löschen wir in der *Account*-Klasse die Zuweisung des *Account*-Objekts an die übergebene *Disbursement*-Instanz:

```
class Account < ActiveRecord::Base
  def book(disbursement)
    self.balance -= disbursement.amount
    disbursement.account = self
  end
end
```

```

    end
  end

```

Mit dem Löschen der *Account*-Referenz aus *Disbursement* und der Anpassung des zugehörigen Tests laufen zwar unsere Unit Tests wieder, allerdings haben wir auch einen Teil der Funktionalität eingebüßt: Das System ist nicht mehr in der Lage, von einem verbuchten Disbursement auf das zugehörige Buchungskonto zu schließen. Diese Funktionalität ist jedoch Teil der Anforderungen und muss daher neu implementiert werden. Damit wir das nicht vergessen, erweitern wir unsere TODO-Liste um zwei neue (Test-)Punkte.

- Prüfen, ob *Account.book* ein neues Booking-Objekt anlegt.
- Test *test.book* erweitern: Überprüfen, ob Booking-Objekt Referenz auf Account besitzt.
- Test *test.book* erweitern: Überprüfen, ob Booking-Objekt Referenz auf Disbursement besitzt.

Die Unit Tests laufen wieder, und eigentlich hätten wir nichts mehr zu tun, wäre da nicht unsere TODO-Liste. Der erste Eintrag erinnert uns daran, dass die Methode *Account.book* für das Anlegen eines neuen *Booking*-Objekts zuständig ist. Das Erzeugen neuer und persistenter Objekte lässt sich am einfachsten testen, indem vor der Ausführung des für die Objekterzeugung verantwortlichen Methodenaufrufs die Anzahl der bereits in der Datenbank existierenden Objekte desselben Typs ermittelt wird. Anschließend erfolgt der Methodenaufruf (in unserem Fall *Account.book*) und eine Prüfung, ob die Anzahl der persistenten Objekte dieses Typs nach dem Aufruf um eins größer geworden ist. Der Test *AccountTest.test_book* wird dafür wie folgt erweitert:

```

class AccountTest < ActiveSupport::TestCase
  fixtures :accounts, :bookings

  def test_book
    disbursement = Disbursement.new
    disbursement.amount = 100.0

    no_bookings = Booking.count
    accounts(:first).book(disbursement)
    assert_equal 2900.0, accounts(:first).balance
    assert_equal no_bookings + 1, Booking.count
  end
end

```

Beachten Sie bitte, dass die neue Version von *AccountTest* ein weiteres Fixture lädt: Sobald Buchungen ins Spiel kommen, muss auch der *AccountTest* zusätzlich das Fixture *bookings* laden.

Die Ausführung des *AccountTests* schlägt fehl, weil die *book*-Methode bisher kein *Booking*-Objekt erzeugt:

```
$ rake test:units
.F..
Finished in 0.054549 seconds.

1) Failure:
test_book(AccountTest)
  [./test/unit/account_test.rb:17:in `test_book'
   /Library/Ruby/Gems/1.8/gems/activerecord-2.0.2/lib/
   active_record/testing/default.rb:7:in `run']:
<3> expected but was
<2>.
```

Schreiben Sie genauso viel Code, wie benötigt wird, um den Test zum Laufen zu bringen. Die einfachste diesbezügliche Möglichkeit ist die Erzeugung und Speicherung eines neuen *Booking*-Objekts in der Methode *Account.book*:

```
class Account < ActiveRecord::Base
  def book(disbursement)
    self.balance -= disbursement.amount
    Booking.create
  end
end
```

Nach dieser Änderung laufen die Tests wieder, und wir können den zugehörigen Eintrag aus der TODO-Liste streichen:

- Prüfen, ob *Account.book* ein neues *Booking*-Objekt anlegt.
- Test *test_book* erweitern: Überprüfen, ob *Booking*-Objekt Referenz auf *Account* besitzt.
- Test *test_book* erweitern: Überprüfen, ob *Booking*-Objekt Referenz auf *Disbursement* besitzt.

Allerdings sind wir noch nicht am Ziel unserer Reise. Ein weiterer Eintrag in der TODO-Liste erinnert uns daran zu überprüfen, ob das erzeugte *Booking*-Objekt nach der Buchung eine Referenz auf das zugehörige Konto besitzt. Dieser Test ist nicht mehr ganz so einfach, da zunächst das neu erzeugte *Booking*-Objekt aus der Datenbank geladen werden muss. Doch welche der drei vorhandenen Buchungen ist die neue? Einfacher wird der Test, wenn man die Methode *Account.book* zunächst so verändert, dass sie das erzeugte *Booking*-Objekt als Rückgabewert liefert.¹

Der Test *book_test* kann jetzt um die Überprüfung der *Account*-Referenz im erzeugten *Booking* erweitert werden:

```
class AccountTest < ActiveSupport::TestCase
  ...
  def test_book
    disbursement = Disbursement.new
    disbursement.amount = 100.0
```

¹ Das Zurückliefern von Werten ist übrigens gang und gäbe in der Ruby-Entwicklung [3].


```

    no_bookings = Booking.count
    booking = accounts(:first).book(disbursement)
    assert_equal 2900.0, accounts(:first).balance
    assert_equal no_bookings + 1, Booking.count
    assert_equal accounts(:first), booking.account
  end
end

```

Der Test schlägt fehl. Also erweitern wir *Account.book* um die vom Test geprüfte *Account-Relation* im erzeugten *Booking-Objekt*:

```

class Account < ActiveRecord::Base
  def book(disbursement)
    self.balance -= disbursement.amount
    Booking.create(:account => self)
  end
end

```

Die Tests laufen wieder, und der zugehörige *TODO-Eintrag* kann gelöscht werden:

- Prüfen, ob *Account.book* ein neues *Booking-Objekt* anlegt.
- Test *test_book* erweitern: Überprüfen, ob *Booking-Objekt* Referenz auf *Account* besitzt.
- Test *test_book* erweitern: Überprüfen, ob *Booking-Objekt* Referenz auf *Disbursement* besitzt.

Als letzter offener Punkt verbleibt die Überprüfung, ob ein neu erzeugtes *Booking-Objekt* eine Referenz auf das zugehörige *Disbursement* besitzt. Die Entwicklung des Tests und der zugehörigen Funktionalität überlassen wir Ihnen. Tipp: Test und Code unterscheiden sich nur unwesentlich von Test und Code der *Account-Referenz* im *Booking*.

Umsetzung der neuen Anforderung

Nachdem das vorausgehende Refactoring das Haushaltsbuch für die neue Anforderung vorbereitet hat, kommen wir jetzt zu deren Umsetzung. Dafür soll die Methode *Account.book* so erweitert werden, dass sie neben einem *Disbursement* optional einen Betrag übergeben bekommt. Ist dieser Betrag vorhanden, so wird nicht mehr der Gesamtbetrag des *Disbursements* verbucht, sondern der übergebene Betrag. Zusätzlich wird der *Disbursement-Betrag* um den übergebenen Wert reduziert. Einen fehlschlagenden Test erhalten wir ganz einfach, indem die Methode *book* mit einem zusätzlichen Betrag aufgerufen wird:

```

class AccountTest < ActiveSupport::TestCase
  ...
  def test_book_with_partial_amount
    disbursement = Disbursement.new
    disbursement.amount = 100.0
  end
end

```

```

    booking = accounts(:first).book(disbursement, 50)
  end
end

```

Die Erweiterung der Methode *book* um einen optionalen Parameter *partial_amount* bringt den Test zum Laufen. Die optionale Natur des Parameters sorgt dafür, dass keiner der existierenden Tests zerbricht.

Als Nächstes wird der Test um eine Assertion erweitert, die sicherstellt, dass nicht mehr der Gesamtbetrag des Disbursements, sondern nur noch der übergebene Teilbetrag verbucht wird:

```

class AccountTest < ActiveSupport::TestCase
  ...
  def test_book_with_partial_amount
    ...
    assert_equal accounts(:first).balance, 2950.0
  end
end

```

Die Methode *book* wird um eine Überprüfung des Parameters *partial_amount* erweitert:

```

class Account < ActiveRecord::Base
  def book(disbursement, partial_amount=nil)
    unless partial_amount.nil?
      self.balance -= partial_amount
    else
      self.balance -= disbursement.amount
    end
    Booking.create(:account => self)
  end
end

```

Jetzt läuft der Test wieder. Als letzter offener Punkt verbleibt die Anforderung, dass der Gesamtbetrag des Disbursements um den verbuchten Teilbetrag reduziert werden muss, in diesem Fall also um 50 Euro. Erweitern Sie dafür zunächst den Test, und passen Sie anschließend die *book*-Methode so an, dass der Test wieder läuft.

14.4.5 Retrospektive

In den vorausgehenden Abschnitten haben wir ein kleines Haushaltsprogramm testgetrieben entwickelt. Das Design der Anwendung erfolgte dabei evolutionär, getrieben durch Tests.

Bemerkenswert ist, dass wir vollständig GUI-frei entwickelt haben. Lästiges und nur mühsam wiederholbares manuelles Testen der Anwendung entfällt. Wir mussten nicht einmal den *index*-View anklicken, um zu testen, ob er wirklich existiert und geöffnet werden kann.

Sie können den GUI-Test jetzt nachholen und überprüfen, ob die Anwendung tatsächlich funktioniert. Starten Sie dazu *script/server*, und geben Sie im Browser

die URL `http://localhost:3000/account/index` ein. Überprüfen und bearbeiten Sie die Liste der erfassten Konten. Legen Sie ggf. vorher die Development-Datenbank an, sofern dies noch nicht geschehen ist.

Wir haben außerdem gelernt, dass vorhandene Tests ein sicheres Refactoring ermöglichen und antreiben können. Dabei ist es wichtig, dass Sie zu jedem Zeitpunkt nur einen Hut aufhaben: den Refactoring- oder den Anforderungshut. In unserem Beispiel war es so, dass wir erst den Refactoring-Hut aufhatten und das System auf die neuen Anforderungen vorbereitet hatten. Anschließend setzten wir den Anforderungshut auf und implementierten die neuen Anforderungen.

Nachdem Sie jetzt den grundsätzlichen Ablauf der testgetriebenen Rails-Entwicklung kennengelernt haben, werden wir in den nächsten Abschnitten die einzelnen Komponenten einer Rails-Anwendung hinsichtlich ihrer Testbarkeit genauer beschreiben.

14.5 Unit Tests – Testen von Modellklassen

Dieser Abschnitt beschreibt die Programmierung von Unit Tests für Modellklassen. Wir beschreiben den grundlegenden Aufbau von Modelltests und geben Ihnen eine Reihe von Testrezepten an die Hand, die Ihnen helfen sollen, typische und wiederkehrende Fragen und Probleme beim Testen von Modellklassen zu lösen.

14.5.1 Struktur und Elemente von Modelltests

Rails generiert für jede Modellklasse eine zugehörige Testklasse mit folgendem Aufbau:

```
class AccountTest < ActiveSupport::TestCase
  fixtures :accounts

  def setup
    @credit = 5000.0
  end

  def teardown
  end

  def test ...

end
```

Modelltests erben von `ActiveSupport::TestCase` und laden über die Klassenmethode `fixtures` die vom Test benötigten Testdaten in die Datenbank. In der Methode `setup` wird das Fixture des Test Cases zusammengestellt, z.B. indem konkrete Datensätze des Datenbank-Fixtures ausgewählt und Instanzvariablen des Tests zugewiesen werden (siehe Listing). Neben der Initialisierung des Test-Fixtures können in der `setup`-Methode weitere, von den Tests benötigte Ressourcen bereitgestellt werden.

Während die *setup*-Methode von Rails generiert wird, muss die Methode *teardown* von Hand hinzugefügt werden, sofern zuvor angeforderte Ressourcen wieder freigegeben werden müssen.

Der *setup*- und *teardown*-Methode folgen die Testmethoden des Test Cases. Die Testmethoden erledigen die eigentliche Arbeit: Sie überprüfen mit Hilfe von Assertions bestimmte Annahmen bezüglich der im Test befindlichen Modellklasse.

14.5.2 Testmethoden

Zunächst stellt sich die Frage, was denn in Bezug auf Modellklassen eigentlich getestet werden muss. Die Antwort kennen Sie bereits: Es muss alles getestet werden, was potenziell schiefgehen oder durch Änderungen zerbrechen kann. Wir haben die Erfahrung gemacht, dass es sinnvoll ist, jede Methode einer Modellklasse (inklusive aller Setter für Attribute) mindestens einmal von einem Test aufrufen zu lassen. Verändert z.B. ein Entwickler das Datenmodell, ohne dass es einen Test gibt, der auf die veränderten Attribute zugreift, wird das Problem im schlimmsten Fall erst von einem Anwender bemerkt, was wir nicht wollen.

Der wesentliche Unterschied zwischen „normalen“ Unit Tests und Unit Tests für Rails-Modellklassen besteht darin, dass ein Modellklassen-Test ein erweitertes Fixture besitzt. Dieses erweiterte Fixture besteht neben den im Test-Setup initialisierten Instanzvariablen aus einem Datenbank-Fixture, d.h. aus einer Menge von Datensätzen, die vor Ausführung jedes Tests frisch in die Datenbank eingefügt werden (siehe dazu Abschnitt 14.5.3).

Damit das funktioniert, sind Modelltests während ihrer Ausführung auf einen laufenden Datenbankserver angewiesen. Diese Tatsache ist ein wesentlicher Unterschied zu „herkömmlichen“ Unit Tests, die externe Systeme, wie Datenbanken, üblicherweise durch Mock-Objekte ersetzen. Rails geht hier den entgegengesetzten Weg und bezieht das externe Datenbanksystem aktiv in den Test mit ein. Das ist auf jeden Fall sinnvoll, weil das Speichern von Domain-Objekten eine der Kernkompetenzen von Modellen ist.

Neben ihrer Kernfunktionalität, der Persistenzlogik für Domain-Objekte, implementieren Modellklassen auch darüber hinausgehend ganz normale Geschäftslogik. Deshalb unterscheiden wir beim Testen von Modellklassen zwischen Tests für Datenbanklogik und Tests für Geschäftslogik.

Tests für die Datenbanklogik stellen im Wesentlichen sicher, dass sich die Instanzen des Modells korrekt laden, speichern und wieder löschen lassen (siehe 14.5.5). Bestehen Abhängigkeiten zwischen Modellinstanzen, muss zusätzlich getestet werden, ob diese Abhängigkeiten korrekt verwaltet werden (siehe 14.5.5).

Die Geschäftslogik einer Modellklasse kann ein breites Spektrum umfassen. Beispiele sind das Inkrementieren von Zählern (z.B. zur automatischen Vergabe von fortlaufenden Buchungsnummern), das Sortieren von Daten oder die Durchführung von komplexen Berechnungen. Aufgrund dieser Vielfalt ist es nicht möglich, eine stringente Vorgehensweise zum Testen von Geschäftslogik zu definieren. Hier gilt es vielmehr, die Kunst des Unit-Testens anzuwenden, d.h. mit Hilfe der von *Test::Unit*

bereitgestellten Standard-Assertions die Erwartungen an die Klasse zu formulieren und zu überprüfen.

Anders hingegen sieht es mit dem Testen der Validierungslogik von Modellklassen aus. Hier haben sich eine Reihe von Testmustern bewährt, die wir Ihnen in Abschnitt 14.5.5 erklären. Bevor wir jedoch in die Details zum Testen der verschiedenen Modellaspekte einsteigen, wollen wir Ihnen eine Einführung in den Aufbau, das Format und die Benutzung der Datenbank-Fixtures von Rails geben.

14.5.3 Testdaten – Fixtures

In Kapitel 13 haben Sie gelernt, dass ein Fixture ein Setup von Testdaten für einen Test Case ist. In diesem Abschnitt werden wir Ihnen die speziellen Rails *Active Record Fixtures* vorstellen.

Ein Active Record Fixture ist eine Datei mit Testdaten für einen bestimmten Active Record-Typ. Jede Modellklasse hat ihr eigenes Fixture, das von Rails bei der Modellerzeugung automatisch generiert wird. Fixtures werden im Verzeichnis *APP_ROOT/test/fixtures* unter dem Pluralnamen der Modellklasse, gefolgt vom Suffix *.yaml*, gespeichert. Alternative Fixture-Formate sind *csv* oder *Single File Fixtures*.

Der Aufruf *ruby script/generate model account* erzeugt neben der Modellklasse *Account* das Fixture *test/fixtures/accounts.yaml*:

```
# Read about fixtures at http://ar.rubyonrails.org/classes/
# Fixtures.html

# one:
#   column: value
#
# two:
#   column: value
first:
  balance: 3000.0
```

Jeder Datensatz eines YAML-Fixtures hat einen Namen. Verwenden Sie statt der vordefinierten auf jeden Fall aussagekräftige Namen, wie z.B. *account_with_credit* oder *account_of.thomas*. Dann fällt das Schreiben und spätere Verstehen der Tests wesentlich leichter.

```
account_with_credit:
  balance: 3000.0
...
```

Dem Namen folgt eine Liste von Attributen mit je einem zugehörigen Wert, getrennt durch einen Doppelpunkt. Die Attributnamen entsprechen den Spaltennamen der zugehörigen Datenbanktabelle.

Verwendung von Fixtures

Ein Test Case, der ein bestimmtes Fixture verwenden will, muss es durch Aufruf der Klassenmethode *fixtures* laden:

```
class AccountTest < ActiveSupport::TestCase
  fixtures :accounts
end
```

Das Laden eines Fixtures bewirkt, dass die darunterliegende Testumgebung vor Ausführung jedes Tests die folgenden Aktionen durchführt:

1. Löschen sämtlicher Datensätze aus der zum Fixture gehörenden Datenbanktabelle.
2. Einfügen aller im Fixture enthaltenen Datensätze in die zugehörige Tabelle.

Bei zunehmender Fixture-Größe kann die permanente Ausführung der oben beschriebenen Schritte zu einer merklichen Zunahme der Ausführungszeiten für Tests führen. Befindet sich in einem Test Case ein Test, der die Daten des Fixtures nicht benötigt, werden die Fixture-Daten trotzdem vor Ausführung des Tests in die Datenbank eingefügt. Diese Tatsache ist einer der Gründe, Tests um Fixtures und nicht um Klassen herum zu organisieren. So werden unnötige Datenbankzugriffe vermieden. Benötigen einzelne Tests einer Testklasse deren Fixtures nicht, ist das ein eindeutiges Indiz dafür, diese Tests in eine andere Testklasse zu verschieben. Eine andere Möglichkeit des Umgangs mit langen Ausführungszeiten aufgrund großer Fixtures ist die Verwendung transaktionaler Fixtures, die wir in Abschnitt 14.5.4 beschreiben.

Auf geladene Fixtures kann innerhalb der Testklasse über eine gleichnamige Getter-Methode zugegriffen werden, die als Parameter den Namen des Fixture-Eintrags als Symbol erwartet. Das folgende Beispiel demonstriert den Zugriff auf das Kontomodell *account_with_credit* im Fixture *accounts.yml*:

```
class AccountTest < ActiveSupport::TestCase
  def test_fixtures
    assert_equal 3000.0, accounts(:account_with_credit).balance
  end
end
```

Häufig benötigt ein Test Daten aus mehreren Fixtures, wie z.B. unsere *AccountTest*-Klasse. Der Test *Account.book* benötigt neben dem Fixture *accounts.yml* das Fixture *bookings.yml*, um nach Durchführung der Buchung zu überprüfen, ob die Datenbank eine neu erzeugte Buchung enthält. Ein Test Case kann mehrere Fixtures laden, indem der Methode *fixtures* eine durch Kommas getrennte Liste der benötigten Fixtures übergeben wird.

Abhängigkeiten von Testdaten

Wenn zu viele Tests dieselben Testdaten verwenden, werden die Abhängigkeiten manchmal zu groß, und es müssen zu viele Tests angepasst werden, bis alles wieder läuft.

Ein neuer Test sollte immer mit der Erstellung der benötigten Testdaten, d.h. mit der Erweiterung der betroffenen Fixtures beginnen. Laufen die vorhandenen Tests mit den neuen Testdaten, ist alles in Ordnung, und Sie können mit der Programmierung der Tests fortfahren. Führt das Bereitstellen neuer Testdaten allerdings schon zum Fehlschlagen einer Reihe vorhandener Tests, so ist dies ein Zeichen dafür, dass die Abhängigkeiten möglicherweise schon zu groß sind. Dann sollten Sie sich überlegen, ob es nicht besser wäre, neue und unabhängige Testdaten zur Verfügung zu stellen.

Fixtures für assoziierte Modelldaten

Selbst in kleineren Projekten haben Sie es beim Schreiben von Tests mit mehreren Modellklassen zu tun, die sich gegenseitig referenzieren. Im richtigen Leben gehört beispielsweise jedes Konto genau einem Kunden, was in ActiveRecord wie folgt modelliert wird:

```
class Account < ActiveRecord::Base
  belongs_to :customer
end
```

In früheren Rails-Versionen konnten Testdaten ausschließlich über die explizite Angabe von Fremdschlüsseln referenziert werden:

```
# test/fixtures/customers.yml
ralf:
  id: 1
  name: Ralf

# test/fixtures/accounts.yml
account_ralf:
  id: 1
  balance: 3000.0
  customer_id: 1
```

Der beschriebene Fremdschlüssel-Ansatz wird schnell unübersichtlich, da ein Blick in *accounts.yml* keinen Aufschluss darüber liefert, welcher Kunde denn genau referenziert wird. Ein wesentlich klarerer Ansatz ist die Verwendung der im Modell deklarierten *belongs_to*-Assoziation innerhalb der Fixture-Datei:

```
# test/fixtures/customers.yml
ralf:
  name: Ralf

# test/fixtures/accounts.yml
account_ralf:
  balance: 3000.0
  customer: ralf
```

Fixture-Einträge referenzieren sich analog zu ActiveRecord-Instanzen über die Angabe des Assoziationsnamens (hier *customer*), gefolgt vom Namen des referenzierten Fixture-Eintrags (hier *ralf*).

Testen Sie Ihre Testdaten

Wir haben die Erfahrung gemacht, dass es in bestimmten Fällen sinnvoll ist, Tests für die Testdaten selbst zu schreiben. Diese Tests stellen das Vorhandensein bestimmter Konstellationen als Voraussetzung für die darauf aufbauenden Tests sicher.

Dazu ein Beispiel: Wir erstellen Testdaten, wonach Benutzer A Mitglied der Gruppe B sein muss. Das Vorhandensein dieser Relation ist Voraussetzung dafür, dass weitere, auf dieser Annahme basierende Tests funktionieren. Zunächst werden die Testdaten um die Relation erweitert. Dann wird ein Test geschrieben, der diese Relation überprüft. Das ist auf jeden Fall sinnvoll, da Fixtures im Nachhinein immer mal wieder angepasst werden. Und da wird schnell mal die Gruppe eines Benutzers geändert, wenn ein neuer Test dies erfordert.

14.5.4 Transaktionale Fixtures

Transaktionale Fixtures nutzen den Transaktionsmechanismus der darunterliegenden Datenbank, um Tests voneinander zu isolieren. Anstatt vor der Ausführung jedes Tests sämtliche Tabellen zu löschen und die Fixture-Daten neu einzufügen, werden transaktionale Fixtures für die gesamte Testklasse ein einziges Mal in die Datenbank geschrieben. Im Anschluss an die Ausführung einer Testmethode wird der ursprüngliche Zustand der Datenbank wiederhergestellt, indem die von der Methode durchgeführten Datenbankoperationen zurückgerollt werden.

Transaktionale Fixtures resultieren in einer deutlichen Zunahme der Ausführungsgeschwindigkeit für Unit Tests und sind seit Rails 1.0 standardmäßig eingeschaltet. Wenn Sie transaktionale Fixtures verwenden wollen, können Sie dies über die in jedem Rails-Projekt enthaltene Datei `test/test_helper.rb` konfigurieren:

```
class Test::Unit::TestCase
  self.use_transactional_fixtures = true
  ...
end
```

Damit transaktionale Fixtures funktionieren, muss die verwendete Datenbank transaktionsfähig sein. MySQL erfüllt diese Anforderung, setzt aber voraus, dass die Tabellen vom Typ *InnoDB* sind. Sie können den Typ einer Tabelle nachträglich ändern:

```
alter table projects type=InnoDB;
```

Ein Nachteil von transaktionalen Fixtures ist, dass bei ihrer Verwendung kein Test des Transaktionsverhaltens der Anwendung mehr möglich ist.

14.5.5 Testrezepte für Modelle

Werden alle Attribute meines Modells korrekt gespeichert?

Für jede Modellklasse sollte ein CRUD-Test existieren. Ein CRUD-Test überprüft, ob die Tabellenstruktur den Erwartungen entspricht und ob die Daten korrekt geladen, gespeichert und wieder gelöscht werden können. Ein CRUD-Test besteht aus folgenden Aktionen:

1. Laden einer Instanz der zu testenden Modellklasse.
2. Vergleich der Attributwerte der Instanz mit denen des zugehörigen Fixtures.
3. Verändern der Attributwerte und Speichern der Instanz.
4. Ggf. neu laden und neu vergleichen.
5. Löschen der Instanz und prüfen, ob die Datenbank einen Satz weniger enthält.

Es muss alles getestet werden, was potenziell schiefgehen kann. Verändert z.B. ein Entwickler das Datenmodell, ohne dass es einen Test gibt, der auf die veränderten oder gelöschten Attribute zugreift, dann wird das Problem im schlechtesten Fall erst von einem Anwender bemerkt.

Für den CRUD-Test bedeutet dies, dass der Test bei jeder Veränderung der Datenbankattribute einer Modellklasse angepasst werden muss. Dazu wird zunächst geprüft, ob das neue bzw. geänderte Attribut den vom zugehörigen Fixture-Eintrag erwarteten Wert besitzt. Ein entsprechend erweiterter Test schlägt zunächst fehl, da weder Fixture noch Datenbank entsprechend angepasst wurden. Anschließend werden Datenbank und Fixture erweitert, und der Test sollte laufen.

Was sich zunächst nach erhöhtem Aufwand anhört, ist bei näherer Betrachtung gar nicht so viel zusätzliche Arbeit. Das hier beschriebene Vorgehen zur Erweiterung von Modellklassen wird sich über kurz oder lang automatisieren.² Als Beispiel soll der Test Case *AccountTest* um einen CRUD-Test für die Klasse *Account* erweitert werden:

```
def test_crud
  # read
  assert_equal 3000.0, accounts(:account_with_credit).balance
  assert_equal "Arbeitsmittel",
    accounts(:account_with_credit).description

  # update
  account = Account.find(3)
  account.balance = 5000.0
  account.description = "Zeitschriften"
  assert account.save
  assert_equal 5000.0, accounts(:account_with_credit).reload.balance
  assert_equal "Zeitschriften",
    accounts(:account_with_credit).description

  # delete
  no_accounts = Account.count
  account.destroy
  assert_equal no_accounts - 1, Account.count
end
```

² Außerdem sparen Sie zusätzlich Zeit, da Sie Ihren Quellcode nicht mehr compilieren und für Ihre Attribute keine Getter und Setter mehr implementieren müssen.

Der Test wird durch zwei Schritte zum Laufen gebracht: die Erweiterung der Tabelle *accounts* um die Spalte *description* und die Erweiterung des Fixtures *accounts.yml* um den Eintrag *description* mit einem passenden Wert, z.B. Arbeitsmittel.

Test von Relationen

Modelle stehen nicht für sich alleine, sondern besitzen Relationen zu anderen Modellen. Z.B. besitzt die zuvor entwickelte *Booking*-Klasse Referenzen auf *Account* und *Disbursement*. Wenn eines von zwei Modellen einer Relation nicht ohne das andere existieren kann, ergibt sich für das Löschen von Modellinstanzen ein weiterer wichtiger Test: Wird ein *Account* gelöscht, muss sichergestellt werden, dass sämtliche *Bookings*, die auf diesen *Account* verweisen, ebenfalls gelöscht werden. Bevor wir einen Test dafür schreiben, erweitern wir das Fixture *bookings.yml* so, dass die erste Buchung auf den ersten *Account* verweist:

```
first_booking:
  account: first
```

Ein Test, der prüft, ob beim Löschen eines *Accounts* auch die den *Account* referenzierenden *Bookings* gelöscht werden, könnte wie folgt aussehen:

```
class AccountTest < ActiveSupport::TestCase
  fixtures :accounts, :bookings
  ...
  def test_remove_bookings
    assert Booking.exists?(bookings(:first_booking).id)
    accounts(:first).destroy
    assert !Booking.exists?(bookings(:first_booking).id)
  end
end
```

Der Test prüft vor dem Löschen des *Accounts*, ob das *Booking* existiert. Anschließend wird der *Account* gelöscht und sowohl für den *Account* als auch für das *Booking* getestet, ob diese auch wirklich aus der Datenbank verschwunden sind.

Wenn Sie für das obige Beispiel eine Datenbank verwenden, in der das Feld *account_id* der Tabelle *bookings* als Fremdschlüssel auf die ID der Tabelle *accounts* definiert wurde, dann funktioniert der neue Test möglicherweise auf Anhieb, wenn Sie für den Fremdschlüssel eine *on_delete*-constraint definiert haben.

Dies ist z.B. unter MySQL möglich. Andere Datenbanken, wie z.B. SQLite, unterstützen keine Fremdschlüssel und erlauben daher auch keine kaskadierenden Löschoperationen (*cascading deletes*). Der Test wird in dieser Umgebung auf jeden Fall scheitern.

In diesem Fall ist es erforderlich, die Abhängigkeiten zwischen Modellen bei der Definition der Modellrelationen in den Modellklassen anzugeben. Dazu greifen wir auf die Option *dependent* zurück, die für die Modellassoziationen *has_one* und *has_many* zur Verfügung steht. Hat die Option den Wert *:destroy* bzw. *:delete_all*, werden alle assoziierten Objekte gelöscht, wenn das Parent-Objekt gelöscht wird.

Was bedeutet das für uns? Richtig, wir müssen die *Account*-Klasse um eine *has_many*-Relation zum *Booking* erweitern und die Option *dependent* auf *:delete_all* setzen. Nach dieser Änderung lässt sich der Test wieder korrekt ausführen:

```
class Account < ActiveRecord::Base
  has_many :bookings, :dependent => :delete_all
  ...
end
```

Funktioniert meine Validierung?

Die Validierungslogik einer Modellklasse lässt sich am einfachsten testen, indem zunächst ein Objekt der zu testenden Modellklasse erzeugt wird und dessen Attribute mit entsprechenden Werten initialisiert werden.³ Nach der Initialisierung des Modellobjekts wird dessen Methode *save* aufgerufen, die *true* oder *false* liefert. Dieser Rückgabewert wird mit *assert* überprüft.

Dazu ein Beispiel: Ein *Booking*-Objekt kann nur gespeichert werden, wenn die Referenzen auf die zugehörigen *Account*- und *Disbursement*-Objekte korrekt gesetzt sind. Der Test für diese Validierung sieht wie folgt aus:

```
class BookingTest < ActiveSupport::TestCase
  ...
  def test_validation
    booking = Booking.new
    assert !booking.save
  end
end
```

Der Test schlägt fehl, da sich das *Booking*-Objekt speichern lässt, obwohl der Test aufgrund einer negativen Validierung das Gegenteil erwartet. Wir bringen den Test zum Laufen, indem wir die *Booking*-Klasse um eine *presence_of*-Validierung für die referenzierte *Account*-Klasse erweitern:

```
class Booking < ActiveRecord::Base
  validates_presence_of :account
end
```

Jetzt läuft der Test. Also erweitern wir den Test um die Zuweisung eines *Account*-Objekts an das zu speichernde *Booking*-Objekt und stellen sicher, dass sich das *Booking*-Objekt immer noch nicht speichern lässt, da die Referenz auf das *Disbursement* noch fehlt:

```
class BookingTest < ActiveSupport::TestCase
  ...
  def test_validation
    booking = Booking.new
    assert !booking.save
  end
end
```

³ Wir umgehen hier die Nutzung des Datenbank-Fixtures bewusst, da die zu validierenden Datenkonstellationen i.d.R. sehr vielfältig sind und sich im Ruby-Code einfacher zusammenstellen bzw. variieren lassen.

```

    booking.account = Account.new
    assert !booking.save
  end
end

```

Und wieder kann das *Booking*-Objekt gespeichert werden, was zur Folge hat, dass der Test erneut fehlschlägt. Es fehlt ja noch die *presence_of*-Validierung für das referenzierte *Disbursement*-Objekt:

```

class Booking < ActiveRecord::Base
  validates_presence_of :account
  validates_presence_of :disbursement
end

```

Schritt für Schritt lässt sich so die Validierung von Modellklassen testgetrieben entwickeln. Zur Sicherheit erweitern wir den Test noch mal, um zu überprüfen, dass sich ein Booking mit referenziertem *Account* und *Disbursement* auch wirklich speichern lässt:

```

class BookingTest < ActiveSupport::TestCase
  ...
  def test_validation
    booking = Booking.new
    assert !booking.save
    booking.account = Account.new
    assert !booking.save
    booking.disbursement = Disbursement.new
    assert booking.save
  end
end

```

Haben Sie grundsätzlich sichergestellt, dass ein Speichern des Objekts nur bei korrekten Daten, d.h. bei positiver Validierung möglich ist, bietet sich als weiterer Test die Überprüfung der erwarteten Fehlermeldungen an. Zu diesem Zweck implementieren wir einen neuen Test *test_validation_errors*.

Der Test überprüft, ob das Fehlerobjekt *errors* für das Attribut *account* eine entsprechende Fehlermeldung enthält, wenn der Buchung kein Konto zugewiesen wurde:

```

class BookingTest < ActiveSupport::TestCase
  ...
  def test_validation
    booking = Booking.new
    booking.save
    assert !booking.errors.empty?
    assert_equal "Eine Buchung muss ein Konto aufweisen",
      booking.errors.on(:account)
  end
end

```

Auch dieser Test schlägt zunächst fehl, da wir der zugefügten Validierung *validates_presence_of* bisher noch keine explizite Fehlermeldung mitgegeben haben. Die folgende Erweiterung von *Disbursement* bringt den Test zum Laufen:

```
class Booking < ActiveRecord::Base
  validates_presence_of :account,
    :message => "Eine Buchung muss ein Konto aufweisen"
  ...
end
```

Erweitern Sie den Test und die Modellklasse um die Überprüfung auf eine entsprechende Meldung bei fehlendem Buchungsbetrag.

Alternativ kann die Validierung einer Modellklasse mit Hilfe der Assertion *assert_valid* getestet werden:

```
def test_validation
  booking = Booking.new
  assert_valid !booking
  booking.account = Account.new
  booking.disbursement = Disbursement.new
  assert_valid booking
end
```

Die Assertion überprüft, ob die übergebene ActiveRecord-Instanz valide ist oder ob deren Validierung fehlschlägt.

Fehlerbehandlung

Modellklassen implementieren Geschäftslogik, bei deren Ausführung es zu Fehlern kommen kann. Fehlerbehandlung ist damit Teil der Logik von Modellklassen, kann zerbrechen und muss getestet werden. Als Beispiel dient die *Account*-Klasse, die eine Ausnahme werfen soll, wenn der zu verbuchende Betrag den aktuellen Kontostand überschreitet. Für die Entwicklung des zugehörigen Tests nutzen wir die Assertion *assert_raise*, die überprüft, ob ein Block eine erwartete Ausnahme wirft:

```
class AccountTest < ActiveSupport::TestCase
  def test_exceptions
    disbursement = Disbursement.new
    disbursement.amount = 3001.0
    assert_raise BalanceExceededException do
      accounts(:first).book(disbursement)
    end
  end
end
```

Die folgende Erweiterung der *book*-Methode bringt den Test zum Laufen:

```
class Account < ActiveRecord::Base
  def book disbursement
    if self.balance < disbursement.amount
      raise BalanceExceededException.new
    end
    ...
  end
end
```

14.6 Funktionale Tests

Testklassen für Controller werden nicht wie Modelltests im Verzeichnis `test/unit`, sondern im Verzeichnis `test/functional` abgelegt. Dies hat damit zu tun, dass Controller-Tests nicht ausschließlich Controller-Funktionalität testen, sondern immer auch die (partielle) Überprüfung der vom Controller genutzten Views mit einbeziehen. Ein Controller-Test testet also nicht nur eine isolierte Einheit des Systems, sondern überprüft das Zusammenspiel von Komponenten in Präsentations- und Controller-Schicht. Und genau dies ist der Grund für die Verwendung des Namens *functional* als Hinweis, dass es sich bei Controller-Tests um funktionale Tests handelt. Ein funktionaler Test ist definiert als ein Test, der die Korrektheit eines Systems über mehrere Ebenen hinweg sicherstellt.

Funktionale Tests beziehen sich in Rails-Applikationen immer auf genau einen Controller. Der Test hat den Namen des zu testenden Controllers, gefolgt vom Wort *Test*, und wird deshalb im Folgenden als Controller-Test bezeichnet. Der Test für den `AccountController` unseres Haushaltsbuchs hat demzufolge den Namen `AccountControllerTest`. Controller-Tests arbeiten nach folgendem Grundmuster:

1. Absetzen eines HTTP-Requests.
2. Test, ob der Request erfolgreich war.
3. Test, ob der Controller die vom View erwarteten Instanzvariablen bereitstellt.
4. Test, ob auf die erwartete Action umgeleitet wird, oder
5. Test, ob der erwartete View ausgeliefert wird.

14.6.1 Testmethoden

Ein Controller-Test beginnt in der Regel mit dem Absetzen eines simulierten HTTP-Requests. Der Testklasse stehen dafür fünf verschiedene Request-Typen zur Verfügung:

- `get`
- `post`
- `put`
- `head`
- `delete`

Jeder Request-Typ wird durch eine gleichnamige Methode abgesetzt. Beispielsweise sendet `get` einen simulierten Get-Request⁴ und füllt das Response-Objekt mit der vom Server gelieferten Antwort:

```
class AccountControllerTest < ActionController::TestCase
  ...
end
```

⁴ Der Webserver muss zur Ausführung der Tests nicht laufen.

```
def test_should_get_index
  get :index
end
end
```

Die Methode `get` erwartet als ersten Parameter die angeforderte Controller-Action, entweder als Symbol oder als String. Optional können der Methode eine Hash mit Request-Parametern und eine Hash mit Session-Werten übergeben werden. Wird der Test ausgeführt, ohne dass die Action vorhanden ist, liefert Rails die Fehlermeldung „`ActionController::UnknownAction: No action responded to index`“, die Sie bereits aus Abschnitt 14.4.2 kennen. Ist die Action vorhanden, läuft der Test durch.

In den Testmethoden einer Controller-Testklasse werden analog zu herkömmlichen Unit Tests bestimmte Annahmen mit Hilfe von Assertions überprüft. Dazu führt Rails eine Reihe neuer Assertions speziell zum Testen von Controllerklassen ein. Mit Hilfe dieser Assertions kann die gesamte Funktionalität von Controllern lückenlos getestet werden. Die häufige Ausrede, wonach etwas aufgrund nicht-testbarer Technologie nicht getestet werden kann, zählt nicht mehr. An ihre Stelle tritt die Chance, ein sich vollständig selbst testendes System zu entwickeln.

14.6.2 Kontrollfluss-Assertions

Controller steuern den Kontrollfluss einer Rails-Anwendung. Ihre Actions liefern Views oder leiten auf andere Actions weiter. Die Kontrollfluss-Funktionalität wird mit Hilfe zweier neuer Assertions getestet:

- `assert_response`
- `assert_redirected_to`

`assert_response`

Die Assertion `assert_response()` überprüft den Status-Code einer Response auf einen vorausgehenden HTTP-Request. Die Assertion erwartet als Parameter einen Response-Typ, der einen einzelnen bzw. einen Bereich von Response-Codes repräsentiert. Die folgenden vier Response-Typen stehen zur Auswahl:

- **:success.** Der Status-Code ist 200, d.h. der Request war erfolgreich.
- **:redirect.** Der Status-Code liegt im Bereich von 300 bis 399, d.h. der Request wurde redirected.
- **:missing.** Der Status-Code ist 404, d.h. die Seite ist nicht vorhanden.
- **:error.** Der Status Code-liegt im Bereich von 500 bis 599, d.h. es ist ein Fehler aufgetreten.

Alternativ zur Angabe des Response-Typs können Sie `assert_response` auch einen einzelnen Wert übergeben: `assert_response(501)`.

Die Verwendungsmöglichkeiten der *assert_response*-Assertion sind vielfältig. Beispielsweise stellt der folgende Test sicher, dass die Action *index* vorhanden ist und erfolgreich aufgerufen werden kann:

```
class AccountControllerTest < ActionController::TestCase
  ...

  def test_should_get_index
    get :index
    assert_response :success
  end
end
```

Der Test stellt nicht nur das Vorhandensein der *index*-Action sicher, sondern überprüft zusätzlich das mit dieser Action assoziierte Template *index.html.erb*. Werden in diesem Template beispielsweise nicht vorhandene Methoden auf Objekten aufgerufen, schlägt der Test mit einer entsprechenden Fehlermeldung fehl. Dies ist ein Beispiel dafür, dass ein funktionaler Test nicht nur den Controller, sondern auch die verwendeten Views testet.

assert_redirected_to

Actions liefern am Ende ihrer Ausführung nicht zwingend ein Template, sondern können auch auf eine andere Action weiterleiten. Testen können Sie diese Funktionalität mit Hilfe der Assertion *assert_redirected_to*. Beispiel: Die Anmeldung eines Benutzers an unserem Buchhaltungssystem schlägt fehl, und die *login*-Action redirected auf die *index*-Action, anstelle das Benutzermenü anzuzeigen. Ein Test für diese Funktion sieht so aus:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_authenticate
    get :authenticate, {"user" => "ralf", "password" => ""}
    assert_redirected_to :action => "index"
  end
end
```

Der Request *authenticate* wird mit einem leeren Passwort abgesetzt, was zu einer fehlschlagenden Anmeldung führt. Dies wird überprüft, indem man mit Hilfe der Assertion *assert_redirected_to* sicherstellt, dass der Request auf die Action *index* redirected wird. Befindet sich die Action in einem anderen Controller, kann dies durch die zusätzliche Angabe des Controller-Namens in der *options*-Hash überprüft werden:

```
assert_redirected_to :controller => "account", :action => "index"
```


14.6.3 Routing-Assertions

Routing ermöglicht die Verwendung von *Pretty URLs* und ist in Abschnitt 5.12 ausführlich beschrieben. Routing kann fehlschlagen und muss getestet werden. Hierfür stehen drei Assertions zur Verfügung:

- `assert_generates`
- `assert_recognizes`
- `assert_routing`

`assert_generates(expected_path, options, [defaults], [extras], [message])`

Die Assertion `assert_generates` stellt sicher, dass sich der erwartete Pfad aus den angegebenen Optionen generieren lässt. Der folgende Test überprüft, ob die erzeugte URL `http://localhost:3000/household` in die erwartete URL `http://localhost:3000/account/list` umgesetzt wird.

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_routing
    assert_generates "household",
      { :controller => "account", :action => "index" }
    ...
  end
end
```

Der Test schlägt fehl, weil das zugehörige Routing bisher nicht konfiguriert wurde. Routen werden in der Datei `APP_ROOT/config/routes.rb` konfiguriert (vgl. Abschnitt 5.12). Die Ergänzung der Datei um die nachfolgende Route bringt den Test zum Laufen:

```
ActionController::Routing::Routes.draw do |map|
  map.connect "household",
    :controller => "account", :action => "index"
  ...
end
```

`assert_recognizes(expected_options, path, [extras])`

Die Assertion `assert_recognizes` stellt sicher, dass der angegebene Pfad korrekt umgesetzt wird und die Optionen dazupassen. Die Assertion ist damit der Umkehrtest zu `assert_generates`:

```
def test_routing
  assert_recognizes({ :controller => "account",
    :action => "index" }, "household")
  ...
end
```

assert_routing(path, options, [defaults], [extras])

Die Assertion *assert_routing* ist ein Wrapper um die beiden vorherigen Assertions. Die Assertion stellt sicher, dass Pfad und Optionen in beide Richtungen passen, d.h. der Pfad lässt sich aus den Optionen generieren (analog zu *assert_generates*), und aus dem Pfad lassen sich die angegebenen Optionen ableiten (analog zu *assert_recognizes*):

```
def test_routing
  assert_routing "household",
    { :controller => "account", :action => "index" }
  ...
end
```

14.6.4 Datencontainer-Assertions

Controller sind nicht ausschließlich für die Steuerung des Kontrollflusses zuständig, sondern speichern, lesen und löschen darüber hinaus Objekte in speziellen Datencontainern, wie der Session oder dem Flash-Speicher. Diese Datencontainer sind als Hashes implementiert, auf die ein Controllertest über entsprechende Getter-Methoden zugreifen kann. Folgende Getter stehen zur Verfügung:

- **assigns()**. Liefert eine Hash mit Instanzvariablen, die von einer Action für einen View bereitgestellt wurden.
- **session()**. Liefert eine Hash mit Objekten, die in der Session gespeichert wurden.
- **flash()**. Hash mit Flash-Objekten, die sich aktuell in der Session befinden.
- **cookies()**. Hash mit den von diesem Request zum Benutzer gesendeten Cookies.

Die Inhalte der Hashes können mit den verschiedenen *Test::Unit Assertions* überprüft werden. Beispielsweise stellt die Assertion *assert_not_nil assigns(:account)* sicher, dass eine ausgeführte Action die Instanzvariable *account* definiert und dem zugehörigen View zur Verfügung stellt. Oder *assert flash.empty?* kann dafür genutzt werden, um sicherzustellen, dass sich keine Objekte im Flash-Speicher befinden.

14.6.5 Template-Assertions

Bisher haben wir nur Assertions zum Testen von Controller-Funktionalität kennengelernt. In diesem Abschnitt wollen wir die so genannten *Template-Assertions* beschreiben, eine Klasse von Assertions, die sicherstellen, dass einem View bestimmte Objekte zur Verfügung stehen.

assert_tag

Mit Hilfe der Assertions *assert_tag* und *assert_no_tag* testen Sie die Struktur und Inhalte Ihrer Views. Die Assertion *assert_tag* erwartet als Parameter den Typ eines HTML-Elements (:tag) und überprüft, ob das Element im Response des vorausge-

henden Requests vorhanden ist. Beispielsweise stellt der folgende Test sicher, dass das Template *edit* ein Eingabefeld für den initialen Kontostand besitzt:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_edit_template
    get :new
    assert_tag :tag => "input", :attributes =>
      { :name => "balance" }
  end
end
```

In der Regel möchten wir nicht nur sicherstellen, dass das Template ein Element eines bestimmten Typs enthält, sondern darüber hinaus prüfen, ob das Element bestimmte Eigenschaften besitzt. Zu diesem Zweck wird der Assertion eine Hash mit optionalen Bedingungen übergeben:

- **:attributes.** Hash mit Attributen, die das HTML-Element enthalten muss
- **:parent.** Parent-Element; muss das HTML-Element besitzen (Hash)
- **:child.** Hash mit Bedingungen, die mindestens eines der direkten Kindelemente der Vorfahren des HTML-Elements erfüllen muss
- **:ancestor.** Hash mit Bedingungen, die mindestens einer der Vorfahren erfüllen muss
- **:descendant.** Hash mit Bedingungen, die mindestens einer der Nachfahren des HTML-Elements erfüllen muss
- **:children.** Hash zum Zählen von Kindelementen
- **:content.** Der Inhalt des Elements entspricht dem angegebenen Wert

Die folgende Test-Methode *test_edit_template* zeigt exemplarisch die verschiedenen Verwendungsmöglichkeiten der *assert_tag()*-Assertion:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_edit_template
    get :new

    # Template enthält ein "input" Feld mit den Namen
    # "balance"
    assert_tag :tag => "input", :attributes => { :name =>
      "balance" }

    # Das input Feld befindet sich in einer Tabelle
    assert_tag :tag => "input", :attributes => { :name =>
      "balance" }, :parent => { :tag => "table" }

    # Die Tabelle enthält ein input Feld
    assert_tag :tag => "table", :child => { :tag => "input" }
```

```

# Die Tabelle befindet sich in einem Formular
assert_tag :tag => "table", :ancestor => {:tag =>
  "form"}

# Das Formular enthält ein input Feld mit Namen
# "balance"
assert_tag :tag => "form", :descendant => {:tag =>
  "input", :attributes => {:name => "balance"}}

# Die Tabelle enthält zwei input Felder
assert_tag :tag => "table", :children => {:count => 2,
  :only => {:tag => "input"}}
end
end

```

assert_no_tag

Die Assertion *assert_no_tag* funktioniert analog zu *assert_tag*, stellt jedoch sicher, dass das angegebene Tag nicht existiert.

assert_template

Die Assertion *assert_template* stellt sicher, dass der zuletzt ausgeführte Request das angegebene Template liefert. Während der Test *test_should_get_index* bisher ausschließlich geprüft hat, ob der *index*-Request erfolgreich war, stellt die erweiterte Version des Tests sicher, dass der Request mit dem Template *index.html.erb* gerendert wird:

```

class AccountControllerTest < ActionController::TestCase
  def test_should_get_index
    get :index
    assert_response :success
    assert_template :index
  end
end

```

14.6.6 Testrezepte für Controller und Views

Enthält meine Session die richtigen Werte?

Ein häufig verwendetes Element von Web-Applikationen ist die HTTP-Session, z.B. zum Speichern von Informationen des jeweils angemeldeten Benutzers (vgl. Abschnitt 5.8). Unit Tests haben über die Collection *session* Zugriff auf die Werte einer Session.

Wollen Sie z.B. eine Action entwickeln, die ein Objekt in die Session schreibt, dann kann ein zugehöriger Test wie folgt aussehen:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_store_into_session
    get :store_into_session, {"key1" => "Hallo", "key2" => "Welt"}
    assert !session.empty?
    assert_equal "Hallo", session[:key1]
  end
end
```

Der Test ruft die Action *store_into_session* mit den Parametern *key1* und *key2* auf und überprüft anschließend, ob die Action den übergebenen Wert in die Session geschrieben hat. Zusätzlich überprüft der Test, ob der in die Session geschriebene Wert dem erwarteten Wert *Hallo* entspricht. Analog zum Speichern in die Session wird das Entfernen von Werten aus der Session getestet:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_remove_from_session
    get :store_into_session, {"key1" => "Hallo", "key2" => "Welt"}
    assert !session.empty?
    get :remove_from_session, {"key1" => "Hallo"}
    assert_nil session[:key1]
  end
end
```

Enthält mein Flash-Speicher die richtigen Werte?

Neben der Session stellt Rails zusätzlich den so genannten Flash-Mechanismus zur Verfügung (vgl. Abschnitt 5.9). Dieser Mechanismus ermöglicht den Transfer von Werten zwischen zwei Requests und eignet sich daher gut zum Anzeigen kurzlebiger Benutzerinformationen, wie z.B. einer Meldung, dass ein Objekt erstellt wurde. Getestet werden Flash-Werte durch Überprüfung der Collection *flash*.

Nehmen wir an, Sie wollen dem Benutzer nach dem Erzeugen eines neuen Kontos eine kurze Meldung in den *index*-View einblenden und für diese Funktion die Flash-Variable nutzen. Wird diese Funktionalität testgetrieben entwickelt, dann sieht der zugehörige Test wie folgt aus:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_create_notice
    get :create
    assert !flash.empty?
    assert_flash_equal "Konto erfolgreich erstellt", "notice"
  end
end
```

Flash-Assertions funktionieren analog zu Session-Assertions: Zuerst wird geprüft, ob die Flash-Variable einen Wert unter einem bestimmten Schlüssel enthält, und dann wird geprüft, ob dieser Wert einem erwarteten Wert entspricht.

Erzeugt mein Controller die vom View erwarteten Instanzvariablen?

Ein Controller stellt seinem View Objekte zur Verfügung, indem er diese Objekte als Instanzvariablen in seinen Actions initialisiert. Testen können Sie diese Funktionalität, indem Sie einen Request auf die jeweilige Action absetzen und anschließend prüfen, ob die Collection *assigns* die vom View erwarteten Objekte enthält.

Als Beispiel dient der *AccountController*, dessen *index*-Action eine Liste aller Accounts laden und in einem zugehörigen View anzeigen soll. Der View iteriert über eine Liste von Konten und benötigt dafür die vom Controller initialisierte Instanzvariable *accounts*:

```
<% @accounts.each do |account| %>
  <%=h account.description %>
  <%= account.balance %>
<% end %>
```

Solange der Controller die Instanzvariable *accounts* nicht instanziiert, würde die Anzeige des Views zur Fehlermeldung „no such method for nil object“ führen. Damit der Fehler aber nicht erst vom Benutzer bemerkt wird, schreiben wir einen Test, der überprüft, ob der View Zugriff auf die Instanzvariable *accounts* hat:

```
class AccountControllerTest < ActionController::TestCase
  ...
  def test_view_has_accounts
    get :index
    assert_not_nil assigns(:accounts)
  end
end
```

Der Test schlägt fehl, solange die Instanzvariable *accounts* in der Action *index* nicht initialisiert wurde, z.B. durch:

```
class AccountController < ApplicationController
  ...
  def index
    @accounts = Account.find(:all)
  end
end
```

Render View

Selbst wenn eine Controller-Action nichts anderes tut, als einen View zu liefern, ist es sinnvoll, einen Test für diese Action zu schreiben. Warum dieser Aufwand? Beim Aufruf einer Action übersetzt der ERb-Prozessor einen RHTML-View in eine HTML-Seite. Greift der View dabei z.B. auf Modellattribute zu, die es nicht gibt, schlägt der Test fehl. Der Test testet also neben der eigentlichen Action, ob der View korrekt programmiert wurde.

Häufig kommt es nämlich vor, dass Sie während der Entwicklungsarbeit Modellattribute verändern, indem Sie diese umbenennen oder löschen (z.B. aus Name wird Vor- und Nachname). Greift ein View jetzt auf die ursprünglichen Attribute zu, dann

wird – sofern kein Unit Test existiert – der Fehler erst bemerkt, wenn der erste Benutzer den View öffnet. Das ist definitiv zu spät.

Wurde mein Template-Objekt korrekt initialisiert?

Für Controller müssen wir testen, ob ein Template nach Ausführung einer Controller-Action auf ein Objekt mit einem bestimmten Namen zugreifen kann (*assert_not_nil assigns(:accounts)*, vgl. Abschnitt 14.6.6).

Manchmal reicht es jedoch nicht aus, einfach nur sicherzustellen, dass ein bestimmtes Objekt im Template zur Verfügung steht, z.B. wenn wir wissen wollen, ob das Objekt bestimmte Eigenschaften hat. Getestet wird dies, indem das in der Collection *assigns* enthaltene Objekt auf die entsprechenden Eigenschaften hin untersucht wird. Beispielsweise erzeugt die *create*-Action ein neues Konto und leitet dann auf den Show-View weiter. Der folgende Test stellt sicher, dass der Show-View Zugriff auf die Instanzvariable *@account* besitzt, welche mit der Kontonummer des neu angelegten Kontos initialisiert wurde:

```
def test_create
  post :create, { :number => "380" }
  assert_equal "380", assigns(:account).no
end
```

Enthält mein View die richtigen Werte?

Manchmal ist es wichtig zu testen, ob ein View bestimmte Werte, z.B. als Einträge in einer Combobox, enthält. In einem System, das wir kürzlich entwickelt haben, erfolgt die Berechnung von Prämien basierend auf einer der Transportarten Land, See und Luft. Der Benutzer wählt eine dieser Transportarten aus einer Combobox aus. Nun hat Thomas die Transportart *Land* gemäß einer Kundenanforderung in *Landtransport* geändert. Da die Prämienberechnung den selektierten Wert aus der Combobox verwendet (select value from rate where location = „Land“), wurden nach dieser Änderung für Landtransporte keine Prämien mehr gefunden.

Einmal abgesehen davon, dass es vielleicht nicht so sinnvoll ist, Geschäftslogik basierend auf String-Werten einer Combobox zu implementieren, hätte ein Test das obige Problem innerhalb kürzester Zeit gefunden, und Thomas hätte es gleich beheben können bzw. müssen, da er nur einchecken darf, wenn alle Unit Tests korrekt laufen. Da wir keinen Test hatten, hat Thomas das Problem nicht sofort bemerkt. Als ich das nächste Mal die Prämienberechnung über die GUI testete, stolperte ich über das Problem und benötigte mehr als eine Stunde für seine Behebung. Jetzt gibt es allerdings einen Test.

Das Beispiel beschreibt eine typische Situation, die uns zwingt, einen Test nach der Implementierung des eigentlichen Features zu schreiben. Immer dann, wenn wir als Entwickler oder unser Kunde ein Problem melden, sollten wir zunächst einen Test schreiben, der das Problem reproduziert. Wie lässt sich das für obiges Problem bewerkstelligen? Richtig, wir überprüfen mit Hilfe der Assertion *assert_tag*, ob der View eine Combobox mit den erwarteten Werten enthält:

```

class CertificateTest < ActionController::TestCase
  ...
  def test_transportart_select_box
    get :new
    assert_tag :tag => "select", :descendant => { :tag =>
      "option", :child => /Land/}
    assert_tag :tag => "select", :descendant => { :tag =>
      "option", :child => /Luft/}
    assert_tag :tag => "select", :descendant => { :tag =>
      "option", :child => /See/}
  end
end

```

Ändert Thomas jetzt die Transportart *See* auf *Seefracht*, dann schlägt die dritte Assertion fehl, und Thomas darf nicht einchecken, solange er das Problem nicht behoben hat.

14.7 Integrationstests

Mit Integrationstests steht seit Rails 1.1 eine dritte Art von Tests zur Verfügung. Während der Fokus von Unit- und Funktionalen-Tests auf dem Test einzelner Modelle bzw. Controller-Actions liegt, stellen Integrationstests sicher, dass die einzelnen Komponenten eines Anwendungsfalls korrekt zusammenspielen. Das System wird also mehr als Ganzes, d.h. von der Oberfläche über die Ausführung einzelner, aufeinanderfolgender Controller-Actions bis hin zur Datenbank getestet. Als Beispiel für einen Integrationstest soll ein Test für den Anwendungsfall „Buchung“ entwickelt werden:

Anwendungsfall: „Buchung“

Ein Benutzer meldet sich am System an, ruft die Funktion „Buchung“ auf und bucht einen Betrag auf ein bestimmtes Konto. Nach erfolgter Buchung meldet sich der Benutzer vom System ab.

Die Entwicklung des Tests beginnt mit der Generierung der Testklasse:

```

$ ruby script/generate integration_test booking_story
  exists  test/integration/
  create  test/integration/booking_story_test.rb

```

In der einfachsten Form besteht ein Integrationstest aus aufeinanderfolgenden Get- und Post-Aufrufen, die den Pageflow des Anwendungsfalls simulieren:

```

class BookingStoryTest < ActionController::IntegrationTest
  fixtures :accounts, :users

  def test_booking
    # Login-Seite anfordern
    get "/user/login"
    assert_response :success
  end
end

```



```
assert_template "user/login"

# User gibt Login und Password ein und submitted die Daten
post "/user/authenticate", :login => "ralf",
                           :password => "stpauli"
assert_response :redirect
follow_redirect!
assert_response :success
assert_template "account/list"

# Buchungsseite anfordern
get "/account/book"
assert_response :success
assert_template "account/book"

# User führt eine Buchung durch
post "/account/book", :account_id => accounts(:first).id,
                     :amount => 100.0
assert_response :redirect
follow_redirect!
assert_response :success
assert_template "account/bookings"
assert_not_nil assigns(:bookings)

# User meldet sich ab
get "/user/logout"
assert_response :success
assert_template "user/goodbye"
end
end
```

Die Methode *test.booking* simuliert die Durchführung eines kompletten Buchungsvorgangs, d.h. der Vorgang wird vom Benutzerlogin über die eigentliche Buchung bis hin zum Benutzerlogout nachgebildet. Im Anschluss an jeden Get- bzw. Post-Aufruf wird die Server-Antwort geprüft (*assert_response*) und sichergestellt, dass das richtige Template gerendert wird (*assert_template*). Im Falle eines Redirects kann dieser mit Hilfe der Methode *follow_redirect!* verfolgt werden, d.h. es wird ein weiterer Request auf die Action des Redirects durchgeführt und die Antwort des Servers entsprechend überprüft.

Integrationstests eignen sich hervorragend für die Automatisierung von Akzeptanztests. Akzeptanztests sind neben Unit Tests ein wesentlicher Bestandteil agiler Entwicklungsmethoden. Beispielsweise schlägt Mike Cohn in seinem Buch *User Stories Applied* [26] vor, für jede User Story (bzw. Anwendungsfall) einen oder mehrere Akzeptanztests zu schreiben, die dann vom Kunden manuell am System durchgeführt werden. Integrationstests sind eine gute Ergänzung zu manuell ausgeführten Akzeptanztests und können diese bis zu einem gewissen Grad sogar ersetzen.

14.7.1 Test-DSLs

Die Einzelschritte der Methode *test_booking* können herausfaktoriert und in einem Modul zusammengefasst werden:

```
module HouseholdTestingDSL
  def go_to_login
    get "/user/login"
    assert_response :success
    assert_template "user/login"
  end

  def enter_username_and_password(who)
    who = users(who)
    post "/user/authenticate", :login => who.login,
                              :password => who.password

    assert_response :redirect
    follow_redirect!
    assert_response :success
    assert_template "account/list"
  end

  def go_to_booking_page
    get "/account/book"
    assert_response :success
    assert_template "account/book"
  end

  def execute_booking
    post "/account/book", :account_id => accounts(:first).id,
                        :amount => 100.0
    assert_response :redirect
    follow_redirect!
    assert_response :success
    assert_template "account/bookings"
    assert_not_nil assigns(:bookings)
  end

  def logout
    get "/user/logout"
    assert_response :success
    assert_template "user/goodbye"
  end
end
```

Nach und nach entsteht so eine auf die jeweilige Anwendung maßgeschneiderte Test-DSL⁵, die die Entwicklung einfacher und ausdrucksstarker Tests erlaubt:

```
class BookingStoryTest < ActionController::IntegrationTest
  include HouseholdTestingDSL
  fixtures :accounts, :users

  def test_booking_version_2
    go_to_login
    enter_username_and_password(:ralf)
    go_to_booking_page
    execute_booking
    logout
  end
end
```

14.7.2 Sessions

Ein weiterer interessanter Aspekt von Integrationstests ist die Tatsache, dass man mit ihnen benutzerspezifische Sessions und somit die Benutzung des Systems durch mehrere Benutzer simulieren kann. 37Signals⁶ verwendet diesen Mechanismus z.B. zum Testen ihrer Campfire-Software und simuliert damit die Interaktion mehrerer gleichzeitig beim System angemeldeter Benutzer.

Eine Integration-Session repräsentiert eine Menge von Requests und Responses, die sequentiell von einem Benutzer ausgeführt wurden.

```
class BookingStoryTest < ActionController::IntegrationTest
  include HouseholdTestingDSL
  fixtures :accounts, :users

  def test_multiple_user_booking
    ralf, thomas = login(:ralf), login(:thomas)

    ralf.go_to_booking_page
    ralf.execute_booking
    assert_equal 3100.0, accounts(:first).reload.balance

    thomas.go_to_booking_page
    thomas.execute_booking
    assert_equal 3200.0, accounts(:first).reload.balance
  end

  private
  def login(who)
    open_session do |sess|
      sess.extend(HouseholdTestingDSL)
```

⁵ Domain Specific Language: eine Programmiersprache, die Anforderungen für bestimmte Problemstellungen besonders gut erfüllt.

⁶ <http://www.37signals.com>

```
    sess.go_to_login
    sess.enter_username_and_password(who)
  end
end
end
```

Der Test wurde um die private Methode *login* erweitert, die mit Hilfe von *open.session* eine neue Integration-Session erzeugt und auf der neu erzeugten Session ein Benutzerlogin ausführt. Das Login wird für die beiden Benutzerobjekte *ralf* und *thomas* aufgerufen, sodass der Test zwei gleichzeitig am System angemeldete Benutzer simuliert.

Die Methode *login* liefert als Ergebnis das erzeugte Session-Objekt. Die neuen Session-Instanzen werden in den beiden lokalen Variablen *ralf* und *thomas* gespeichert, und die folgenden Aufrufe von *go_to_booking_page* und *execute_booking* werden nicht mehr auf einer allgemeingültigen Session, sondern auf den benutzerspezifischen Integration-Sessions ausgeführt.

14.8 Testen von E-Mails

Zu Beginn dieses Kapitels erwähnten wir, dass sich nahezu jeder Aspekt einer Rails-Anwendung durch einen Unit Test absichern lässt. Der Fokus dieses Kapitels lag auf der Entwicklung von Tests für Modelle, Controller und Views. Darüber hinaus enthält Rails weitere APIs zum Testen der E-Mail- und Web Service-Funktionalität einer Anwendung. Eine Anleitung zum Testen von E-Mail-Funktionalität finden Sie in Abschnitt 9.1.4.

14.9 Mock-Objekte

Ein Mock-Objekt ist ein Stellvertreter, das ein externes System während der Ausführung von Unit Tests ersetzt. Angenommen, ein auf Basis von Rails entwickeltes Shop-System ermöglicht dem Benutzer die Bezahlung einer Bestellung per Kreditkarte. Die Anwendung muss also die vom Anwender bereitgestellten Kreditkarteninformationen sowie den Bestellwert an ein angeschlossenes Kreditkarteninstitut übertragen. Der zugehörige Controller könnte wie folgt aussehen:

```
class PaymentController < ActionController::Base
  def credit_card_payment
    CreditCardGateway.new.do_payment(params[:number], ...)
  end
end
```

In unserer Test-Umgebung steht das Kreditkarteninstitut für Testzwecke nicht zur Verfügung, was ein Problem darstellt. Außerdem sollen Fehler des Instituts unsere Testergebnisse nicht verfälschen, da wir unser System und nicht die Funktion des Kreditkarteninstituts testen wollen – ein klassisches Beispiel für den Einsatz von Mock-Objekten.

Die Mock-Objekte einer Rails-Anwendung liegen im Verzeichnis *test/mocks/test*. Bei der Ausführung von Unit Tests prüft Rails das Mock-Verzeichnis auf gleichnamige Dateien und instanziiert das Mock-Objekt anstelle des richtigen Objekts. Alles, was wir tun müssen, ist, die eine Datei *credit_card_gateway.rb* im Mock-Verzeichnis *test/mocks/test* anzulegen:

Listing 14.1: *test/mocks/test/credit_card_gateway.rb*

```
class CreditCardGateway
  def do_payment
    true
  end
end
```

Im Unit Test instanziiert Rails das Mock-Gateway und in der Produktionsumgebung das „richtige“ Gateway.

Mock-Objekte müssen nur Methoden implementieren, die auch wirklich aufgerufen werden. Das „richtige“ Gateway im obigen Beispiel besitzt ganz sicher deutlich mehr als die von unserem Mock implementierte *do_payment*-Methode. Weil wir aber wissen, dass unser Controller nur diese eine Methode benötigt, ist es völlig ausreichend, auch nur diese Methode zu definieren.

14.10 Zusammenfassung

Auch testgetrieben entwickelte Software ist nicht fehlerfrei. Wird ein Fehler im Akzeptanztest gefunden, ist es sinnvoll, auch für diesen Fehler zunächst einen fehlschlagenden Test zu schreiben.

Unit Tests sind nur ein Teil der Miete im Testgeschäft. Eine andere Form von Tests sind Akzeptanztests. Im Java-Umfeld etabliert sich hier z.B. das Werkzeug FIT.⁷ Es gilt zu evaluieren, ob wir FIT auch im Rails-Umfeld nutzen können.

Sie werden feststellen, dass Sie bald mehr Test-Code als Produktions-Code entwickeln werden. Das ist gut so, denn Rails erhebt ja gerade den Anspruch, mehr Funktionalität mit weniger Code umzusetzen. Die hierdurch gesparte Zeit können wir für die Entwicklung eines feinmaschigen Testnetzwerks nutzen, um so wartbare und langlebige Systeme zu bauen.

⁷ <http://fit.c2.com/>

14.11 Assertions – Übersicht

Tabelle 14.1 enthält eine Zusammenfassung der Rails-spezifischen Assertions.

Tabelle 14.1: Ruby on Rails-Assertions

Assertion
<code>assert_difference(klass.method, diff, block)</code> Stellt sicher, dass ein Aufruf der Methode <i>method</i> auf der Klasse <i>klass</i> vor und nach der Ausführung des Blocks eine Differenz von <i>diff</i> liefert.
<code>assert_no_difference(klass.method, block)</code> Stellt sicher, dass ein Aufruf der Methode <i>method</i> auf der Klasse <i>klass</i> vor und nach der Ausführung des Blocks keine Differenz liefert.
<code>assert_response(type, [message])</code> Stellt sicher, dass der Status-Code des letzten Requests dem angegebenen Typ entspricht.
<code>assert_redirected_to(options={}, [msg])</code> Stellt sicher, dass die zuletzt ausgeführte Action zu der in den Optionen spezifizierten Action redirected wird.
<code>assert_generates(expected_path, options, [defaults], [extras], [message])</code> Stellt sicher, dass der Pfad aus den angegebenen Optionen generiert werden kann.
<code>assert_routing(path, options, [defaults], [extras])</code> Stellt sicher, dass Pfad und Optionen in beide Richtungen voneinander abgeleitet werden können.
<code>assert_recognizes(expected_options, path, [extras])</code> Stellt sicher, dass der angegebene Pfad korrekt umgesetzt wird und die Optionen dazupassen.
<code>assert_tag(options)</code> Stellt sicher, dass der Response des letzten Requests das in den Optionen angegebene Element enthält und den spezifizierten Attributen genügt.
<code>assert_select(selector, equality?, message?)</code> <code>assert_select(element, selector, equality?, message?)</code> Die Assertion verwendet die Syntax von CSS-Selektoren, um sicherzustellen, dass eine HTML-Datei bestimmte Elemente enthält.
<code>assert_no_tag(options)</code> Stellt sicher, dass der Response des letzten Requests das in den Optionen angegebene Element nicht enthält.
<code>assert_template(expected)</code> Stellt sicher, dass der zuletzt ausgeführte Request mit dem angegebenen Template gerendert wird.

Anhang

Der Anhang beschreibt einige wichtige Konfigurationsparameter der einzelnen Rails-Komponenten. Die Parameter werden entweder in der zentralen Umgebungsdatei `config/environment.rb` oder in den umgebungsspezifischen Dateien im Verzeichnis `config/environments` gesetzt.

Active Record

ActiveRecord::Base.colorize_logging = true | false

Diese Option ist standardmäßig *true* und sorgt für farbiges Logging im Terminal-Fenster.

ActiveRecord::Base.logger = Logger-Objekt

Der von Active Record verwendete Logger. In der Default-Konfiguration wird ein Standard-Ruby-Logger erzeugt, der seine Ausgaben in die Log-Datei der jeweiligen Umgebung schreibt.

ActiveRecord::Base.pluralize_table_names = true | false

Gibt an, ob der Modellname bei der Ermittlung der zugehörigen Tabelle pluralisiert wird.

ActiveRecord::Locking.lock_optimistically = true | false

Schaltet optimistisches Locking an oder aus.

ActiveRecord::Timestamp.record_timestamps = true | false

Active Record aktualisiert beim Speichern von Modellinstanzen automatisch die Felder mit den Namen *created_at*, *created_on*, *updated_at*, *updated_on*. Die automatische Aktualisierung wird mit dieser Option an- oder ausgeschaltet.

Action Controller

ActionController::Base.consider_all_requests_local = true | false

Ist diese Option *true*, werden alle Requests als lokale Requests interpretiert, was dazu führt, dass im Fehlerfall ein View mit Stacktrace und Kontextinformationen ausgeliefert wird.

ActionController::Base.debug_routes = true | false

Diese Option liefert detaillierte Hinweise über das Routing der Anwendung und eignet sich insbesondere fürs Debugging der Routing-Einstellungen.

ActionController::Base.logger = Logger-Objekt

Der von Controllern verwendete Logger. In der Default-Konfiguration wird ein Standard-Ruby-Logger erzeugt, der seine Ausgaben in die Log-Datei der jeweiligen Umgebung schreibt.

ActionController::Base.template_root = Verzeichnis

Basisverzeichnis für Views. Der Default-Wert ist `app/views`.

ActionController::Base.perform_caching = true | false

Schaltet das Caching an oder aus.

ActionController::Base.fragment_cache_store = Caching-Klasse

Für das Fragment-Caching verwendeter Mechanismus (vgl. Abschnitt 6.9.3).

ActionController::Base.ignore_missing_templates = true | false

Ist diese Option *true*, wirft Rails keine Ausnahme, wenn das Template einer Action unauffindbar ist. Die Option ist standardmäßig *false*.

Datenbankmigration

Migrationsskript erzeugen

Ein Migrationsskript wird automatisch bei der Generierung eines Modells erzeugt. Dieses Verhalten kann durch die Angabe der Option `-skip-migration` unterbunden werden. Ein Migrationsskript ohne Modell wird direkt per Generator erzeugt, z.B:

```
$ ruby script/generate migration add_table_projects
exists db/migrate
create db/migrate/003_add_table_projects.rb
```

Migrationsskript ausführen

Ein Migrationsskript wird über *rake* wie folgt ausgeführt:

```
$ rake db:migrate
$ rake db:migrate VERSION=6
```

Schema-Befehle im Migrationsskript

Es stehen folgende Befehle zur Bearbeitung eines Schemas zur Verfügung. Darüber hinaus können im Skript beliebige Befehle ausgeführt werden. Achten Sie darauf, alle Änderungen in *up()* und *down()* wieder rückgängig zu machen!

create_table(name, options)

Erzeugt eine Tabelle namens *name*. Über die Hash *options* können SQL-Optionen wie z.B. *DEFAULT CHARSET=UTF-8* hinzugefügt werden.

```
def self.up
  create_table(:projects,
    :options => "DEFAULT CHARSET=UTF-8 ENGINE=InnoDB")
  ...
end
```

Der Primärschlüssel ist per Default *id* und kann über *:primary_key => „myid“* geändert werden. Eine temporäre Tabelle wird über *:temporary => true* definiert.

drop_table(name)

Löscht die Tabelle *table_name* und damit alle Daten.

```
def self.down
  drop_table :projects
end
```

rename_table(old_name, new_name)

Benennt die Tabelle *old_name* in *new_name* um.

```
def self.up
  rename_table :projects, :web_projects
end
```

add_column(table_name, column_name, type, options)

Fügt eine neue Spalte *column_name* vom Typ *type* der Tabelle *table_name* hinzu. Der Typ kann einer der folgenden sein: *:string*, *:text*, *:integer*, *:float*, *:datetime*, *:timestamp*, *:time*, *:date*, *:binary*, *:boolean*. Über die Optionen können das Limit, der Defaultwert und *NULL* erlaubt oder nicht definiert werden:

```
def self.up
  add_column :projects, :name, :string,
    {:limit => 16, :null=>false, :default=>""}
  add_column :price, :float,
    :limit => "10,3", :null => false, :default => 0.0
end
```

remove_column(table_name, column_name)

Löscht die Spalte *column_name* aus der Tabelle *table_name*.

```
def self.up
  remove_column :projects, :name
end
```

rename_column(table_name, column_name, new_column_name)

Benennt die Spalte *column_name* in *new_column_name* in der Tabelle *table_name* um. Typ und Inhalt bleiben gleich.

```
def self.up
  rename_column :projects, :name, :project_name
end
```

change_column(table_name, column_name, type, options)

Ändert die Spalte *column_name* der Tabelle *table_name* auf den Typ *type*. Die Optionen sind analog zu *add_column*.

```
def self.up
  change_column :projects, :name, :string, :limit => "32"
end
```

change_column_default(table_name, column_name, default)

Ändert den Defaultwert der Spalte *column_name* der Tabelle *table_name* auf den Wert *default*.

```
def self.up
  change_column_default :projects, :name, "rails"
end
```

add_index(table_name, column_name, index_type)

Fügt den Index *column_name* vom Typ *index_type* der Tabelle *table_name* zu. Für *index_type* kann auch eine Hash angegeben werden, die neben dem Typ auch den Namen des Index definiert.

```
def self.up
  add_index :projects, :name, "UNIQUE"
  add_index :projects, :name, {:name => "myidx", :unique => true}
end
```

remove_index(table_name, column_name)

Löscht den Index *column_name* auf der Tabelle *table_name*.

```
def self.up
  remove_index :projects, :name
  remove_index :projects, :myidx
end
```

Zugriff über Modelle

Sind die Modelle bereits erzeugt, kann über sie auf die zugehörige Tabelle im Skript zugegriffen werden, z.B.:

```
def self.up
  ...
  add_column :projects, :department, :string
  ...
  Project.find(:all).each do |p|
    p.update_attribute(:department, ...)
  end
end
```

Direkt SQL ausführen

Um SQL direkt im Skript auszuführen, wird die Methode `execute()` verwendet:

```
def self.up
  execute("alter table projects auto_increment = 100400")
end
```

Action Mailer

ActionMailer::Base.default_charset = String

Der Zeichensatz der E-Mail, z.B. „iso-8859-1“. Default ist „utf-8“.

ActionMailer::Base.default_content_type = String

Der default Content-Type der E-Mail, Default ist „text/plain“.

ActionMailer::Base.delivery_method = :smtp | :sendmail | :test

Die Versandart über SMTP-Server, Sendmail oder simuliert im Testbetrieb.

ActionMailer::Base.logger = Logger

Logger zum Schreiben von Meldungen, per Default `nil`.

ActionMailer::Base.perform_deliveries = true | false

Durch die Option `false` kann der Versand von E-Mails beim Aufruf von `deliver_XXX` unterbunden werden. Dies kann bei Tests hilfreich sein.

ActionMailer::Base.template_root = String

Pfad auf das Basisverzeichnis der Templates für die E-Mail-Texte, Default ist `app/views`.

ActionMailer::Base.raise_delivery_errors = true | false

Beim E-Mail-Versand im Fehlerfall eine Ausnahme werfen oder nicht.

ActionMailer::Base.smtp_settings = Hash

Verbindungsparameter zum SMTP-Server:

- ***:address***
Die Domain des Servers, z.B. „smtp.web.de“.
- ***:port***
Der Port des Servers, bei POP3 häufig 25.
- ***:domain***
Identifikation des Client-Rechners, z.B. `www.domain.de`
- ***:authentication***
Authentifizierungsart: `:plain`, `:login` oder `:cram_md5`.⁸

⁸ Siehe <http://de.wikipedia.org/wiki/Cram-md5>

- `:user_name`
Der Benutzername für das Login.
- `:password`
Das Passwort für das Login.

Mail-Attribute

subject = String

Der Betreff der E-Mail, z.B. `@subject = „News: RubyOnRails.info“`.

recipients = String | Array

Eine oder mehrere E-Mail-Adressen des oder der Empfänger, z.B. `@recipients = [„ralf@b-simple.de“, „Thomas Baustert <thomas@b-simple.de>“]`.

cc, bcc= String | Array

Eine oder mehrere E-Mail-Adressen als Carbon oder Blind Carbon Copy, z.B. `@cc = „copy@b-simple.de“`.

from = String

Die E-Mail-Adresse des Absenders, z.B. `@from = „news@b-simple.de“`.

body = Hash

Das Attribut wird nicht direkt in der E-Mail verwendet, sondern ist für die Übergabe von Werten aus der Anwendung an das Template zuständig, das den Text der Mail definiert. Die Werte in der Hash stehen im Template über Instanzvariablen zur Verfügung. Der Name der Instanzvariablen ist gleich dem Schlüssel in der Hash, z.B. `:project => rails_project` in der Hash und `@project` im Template.

content_type = String

Über das Attribut wird die Art des E-Mail-Inhalts definiert, z.B. `@content_type = „text/html“`.

headers = Hash

Im Attribut `@headers` werden weitere Parameter für die Kopfdaten der Mail bestimmt, z.B. `@headers = { 'reply-to' => „admin@b-simple.de“ }`.

charset = String

Hiermit wird der Zeichensatz der E-Mail definiert. Default ist „utf-8“ oder der Eintrag in `ActionMailer::Base.default_charset`.

sent_on = Time

Der Zeitpunkt der E-Mail-Erstellung, z.B. `@sent_on = Time.now`.

implicit_parts_order = Array

Wenn die implizit über Templates zusammengefügte E-Mail aus mehreren Teilen besteht, kontrolliert diese Variable die Sortierung der Teile. Default ist dabei die Reihenfolge [„text/html“, „text/enriched“, „text/plain“].

Multipart Attribute

Bei einer Multipart können pro Teil (Part) folgende Attribute über entsprechende Setter gesetzt werden:

body()

Der Inhalt, z.B. ein Bild oder eine PDF-Datei.

charset()

Der Zeichensatz, z.B. „UTF-8“ oder „ISO-8859-1“.

content_disposition()

Die Anordnung beim Versand: „inline“(innerhalb der E-Mail) oder „attachment“(als Anhang).

content_type()

Der Inhaltstyp, z.B. „text/plain“, „application/pdf“ oder „image/gif“.

filename()

Der beim Speichern vorgeschlagene Dateiname, z.B. „report.pdf“.

headers()

Weitere Kopfdaten des Teils als Hash, z.B. { :sent_on => Time.now }.

transfer_encoding()

Setzt die Verschlüsselung des Teils. Gültige Werte sind: „quoted-printable“, „base64“, „7bit“, „8bit“, „binary“.

Literaturverzeichnis

- [1] *Kent Beck*: Test Driven Development by Example, Addison-Wesley Professional, 2002
- [2] *Johannes Link*: Softwaretests mit JUnit, dpunkt.verlag, 2005
- [3] *Dave Thomas*: Programming Ruby, Second Edition, Pragmatic Bookshelf, 2005
- [4] *Dave Thomas, Andy Hunt*: Der Pragmatische Programmierer, Hanser, 2004
- [5] *Michael Clark*: Projekt-Automatisierung, Hanser, 2005
- [6] *Mike Beedle, Ken Schwaber*: Agile Software Development with Scrum, Prentice Hall, 2001
- [7] *Martin Fowler*: Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002
- [8] *Justin Gehtland, Ben Galbraight, Dion Almaer*: Pragmatic Ajax, Pragmatic Bookshelf, 2006
- [9] *Frank Westphal*: Testgetriebene Entwicklung mit JUnit & FIT, dpunkt.verlag, 2005.
- [10] *Leonard Richardson und Sam Ruby*: RESTful Web Services, O'Reilly, 2007
- [11] *Michael Schürig*: <http://wiki.rubyonrails.com/rails/show/OverridingRailsMessagesInAnotherLanguage>
- [12] Programming Ruby Online:
<http://www.ruby-doc.org/docs/ProgrammingRuby>
- [13] Rails Homepage: <http://www.rubyonrails.org>
- [14] Rails API: <http://api.rubyonrails.org>
- [15] JavaScript Prototype-Bibliothek: <http://prototype.conio.net>
- [16] JavaScript-Bibliothek: <http://script.aculo.us>
- [17] RubyGem Online-Dokumentation: <http://docs.rubygems.org>

- [18] Ruby-Bibliothek zur Internationalisierung: <http://ri18n.berlios.de>
- [19] JUnit Test Infected: Programmers Love Writing Tests:
<http://junit.sourceforge.net/doc/testinfected/testing.htm>
- [20] Dynamic Typing: <http://www.martinfowler.com/bliki/DynamicTyping.html>
- [21] Ajax: A New Approach to Web Applications: <http://www.adaptivepath.com/publications/essays/archives/000385.php>
- [22] *Cody Fauser*: PDF: RJS Templates for Rails, O'Reilly, 2006
- [23] Subversion Homepage: <http://subversion.tigris.org>
- [24] Subversion Online Handbuch: <http://svnbook.red-bean.com>
- [25] *Mike Mason*: Pragmatic Version Control Using Subversion, Pragmatic Bookshelf, 2005
- [26] *Mike Cohn*: User Stories Applied, Addison-Wesley, 2004
- [27] *Chad Fowler*: Rails Recipes, The Pragmatic Bookshelf, 2006
- [28] *Roy Fielding*: Architectural Styles and the Design of Network-based Software Architectures, <http://www.ics.uci.edu/fielding/pubs/dissertation/top.htm>

Stichwortverzeichnis

- :after 274
- :before 274
- :complete 274
- :conditions 68
- :dependent 81, 87
- :failure 274
- :include 72
- :interactive 274
- :joins 72
- :limit 71
- :loaded 274
- :loading 274
- :offset 71
- :order 71
- :polymorphic 96
- :success 274
- :through 97

Überwachung 306

1:1-Beziehungen 78

1:N-Beziehungen 84

404.html 293

422.html 293

500.html 293

Action Controller 121

Actions 122

after_filter() 141

around_filter() 142

Base 121

Base.fragment_cache_store 192

Base.template_root 124

before_filter() 141

Caching aktivieren 187

Caching::Fragments 192

Caching::Sweeper 189

Datei-Download 156

Datenaustausch 126

DoubleRenderError 126

Filter 141

Filterklassen 144

flash() 140

Flash-Speicher 140

Generator 121

hide_action() 123

Inline-Filter 144

Instanzvariablen 126

layout() 154

Layouts 153

Massenzuweisung 128

Massenzuweisung schützen 129

Mehrdimensionale Parameter 128

Named Routes 150

params 127

prepend_after_filter() 145

prepend_before_filter() 145

private Methoden 123

process() 148

protected Methoden 123

redirect_to() 131

Redirects 130

Reflection 122

render() 124

Request Parameter 127

reset_session() 139

Responses 123

Root Routes 150

Routing 147

send_data() 156

- send_file() 157
- URL-Generierung 152
- URL-Parameter 122
- url_for() 152
- View-Verzeichnis 124
- Action Mailer 249
 - attachment, Methode 260
 - Base.smtp_settings 255, 256
 - Base64 259
 - create_XXX 253
 - deliver_XXX 253
 - E-Mail versenden 249
 - E-Mail-Attribute 250
 - Fehler beim Versand 255
 - Fixture 256
 - Helper Modul 252
 - Helper verwenden 251
 - implicit_parts_order 260
 - Mail, eingehende verarbeiten 261
 - Multipart explizit 258
 - Multipart implizit 260
 - Multipart Mails 258
 - part, Methode 259
 - Partial Views verwenden 253
 - PDF versenden 258
 - POP3, Abruf von 261
 - receive, Methode 261
 - Sendmail, Versand über 255
 - SMTP, Versand über 253, 255
 - Templates 250
 - Test, E-Mail-Versand 256
 - Test, Empfang einer E-Mail 262
 - Testumgebung 257
 - TMail 253
- Action View 161
 - Anzeige Validierungsfehler 185
 - application.html.erb 180
 - ApplicationHelper 164
 - Auswahlboxen 173
 - auto_link() 172
 - button_to() 176
 - capture() 178
 - checkbox() 169
 - content_for() 178
 - content_tag() 171
 - date_select() 172
 - Debugging 179
 - error_message_on() 185
 - error_messages_for() 185
 - fields_for() 167
 - form_for() 165
 - FormHelper 169
 - Formular 168
 - h() 179
 - Helper-Modul 163
 - hidden_field() 170
 - HTML-Code filtern 179
 - HTML-Formular 168
 - HTML-Template 161
 - html_escape() 179
 - human_size() 172
 - image_tag() 178
 - Instanzvariablen 163, 180
 - javascript_include_tag() 177
 - Layouts 180
 - link_to() 175
 - mail_to 177
 - number_with_delimiter() 171
 - options_for_select() 174
 - options_from_collection_for_select() 175
 - Partials View 181
 - password_field_tag() 170
 - radio_button() 169
 - radio_button_tag() 170
 - sanitize() 179
 - select() 174
 - select_tag() 171
 - strip_links() 172
 - stylesheet_link_tag() 177
 - tag() 171
 - Template, HTML 161
 - Template, XML 186
 - text_area() 169
 - text_field() 168
 - text_field_tag() 170
 - truncate() 172
 - Versteckte Felder 170
 - View 161
 - XML-Template 186
 - yield 180
- ActionController

- protect_from_forgery() 167
- session() 138
- verify() 133
- Actions 122
- ActionController::InvalidAuthenticityToken 167
- ActionView
 - authenticity_token() 167
- Active Record 57
 - 1:1-Beziehungen 78
 - 1:N-Beziehungen 84
 - acts_as_list() 106
 - acts_as_tree() 108
 - after_create() 115
 - after_find() 115
 - after_initialize() 115
 - after_save() 115
 - after_update() 115
 - after_validation() 115
 - after_validation_on_create() 115
 - after_validation_on_update() 115
 - Assoziationen 76
 - Attribute 58
 - before_create() 115
 - before_save() 115
 - before_update() 115
 - before_validation() 115
 - before_validation_on_create() 115
 - before_validation_on_update() 115
 - belongs_to() 77, 82, 87
- Callbacks 68, 115
 - Aufrufreihenfolge 115
- column_names() 75
- columns() 75
- content_columns() 75
- create() 65
- delete() 68
- delete_all() 68
- destroy() 68
- destroy_all() 68
- deutsche Modell- und Tabellennamen 62
- direkt verwenden 63
- errors() 110
- establish_connection() 63
- find() 65
- find_all_by() 73
- find_by() 73
- find_by_sql() 74
- Finder 68
- Fremdschlüssel 76
- has_and_belongs_to_many() 90
- has_many() 76, 85
- has_one() 78
- Komplexe Modellattribute 59
- Konkurrierende Zugriffe 119
- Konstruktor 65
- Locking 119
- Metadaten 75
- N:M-Beziehungen 70
- Objekte erzeugen 65
- Objekte löschen 68
- Objekte laden 65
- observe() 118
- Observer 118
- Pattern 58
- Primärschlüssel 60
- reload() 66
- save() 65, 66
- save_with_validation() 110
- self 58
- set_primary_key 60
- set_table_name() 61
- Transaktionen 103
- update() 67
- update_all() 67
- update_attribute() 67
- update_attributes() 67
- validate() 110
- validate_on_create() 111
- validate_on_update() 111
- validates_acceptance_of() 112
- validates_associated() 112
- validates_confirmation_of() 112
- validates_each() 113
- validates_exclusion_of() 113
- validates_format_of() 113
- validates_inclusion_of() 113
- validates_length_of() 114
- validates_numericality_of() 115
- validates_presence_of() 111
- validates_uniqueness_of() 115

- Validierung 110
- Vererbung 100
- ActiveResource 225
- ActiveSupport::TestCase 362
 - setup() 362
 - teardown() 362
- acts_as_list() 106
 - :order 107
 - position 106
- acts_as_tree() 108
 - :foreign_key 108
 - :order 108
 - parent_id 108
- after_create() 115
- after_filter() 141
- after_find() 115
- after_initialize() 115
- after_save() 115, 117
- after_update() 115
- after_validation() 115
- after_validation_on_create() 115
- after_validation_on_update() 115
- Aggregation 99
 - composed_of() 99
- Ajax 263
 - application.js 265
 - auto_complete_for() 282
 - Autocompletion 281
 - Callback-Methoden 274
 - :after 274
 - :before 274
 - :complete 274
 - :failure 274
 - :interactive 274
 - :loaded 274
 - :loading 274
 - :success 274
 - controls.js 265
 - Debugging 289
 - dragdrop.js 265, 275
 - draggable_element() 275
 - drop_receiving_element()
 - drop_receiving_element() 275
 - Dynamische Suchlisten 270
 - effects.js 265
 - FireBug 289
 - form_remote_tag() 267, 269
 - JavaScript 270
 - JavaScriptGenerator 284
 - observe_field() 270
 - page-Objekt 284
 - Partial-Views für Listen 269
 - Prototype 265
 - prototype.js 265
 - render rjs 283
 - RJS Class Proxy 287
 - RJS Collection Proxy 288
 - RJS Element Proxy 288
 - RJS im Controller 286
 - RJS Templates 282
 - RJS und Helper-Methoden 286
 - RJS und Redirect 286
 - Scriptaculous 265
 - Task-Schnellerfassung 267
 - Testen 289
 - Testen mit ARTS 289
 - text_field_with_auto_complete() 281
 - und Rails 265
 - update_page() 284
 - visual_effect 285
- Ajax-Engine 264
- Akzeptanztests 384
- Alternative Webserver 298
- Anwendung warten 305
- Apache und Mongrel 295
- application.html.erb 180
- application.js 265
- ApplicationHelper-Modul 164
- around_filter() 142
- assert_raise() 372
- assert_tag() 354
- assert_valid() 372
- Assertions – Übersicht 389
- assigns 354
- Assoziationen 76
- attr_accessible() 129
- attr_protected() 129
- Ausnahmen
 - rescue_from 136
- Auswahlboxen 173
- Authentication per HTTP 146
- authenticity_token() 167

- auto_complete_for() 282
- auto_increment 60
- auto_link() 172
- Autocompletion 281
- Bäume 105
- Backup 306
- before_create() 115
- before_destroy() 116
- before_filter() 141
 - :except 143
 - :only 143
- before_save() 115
- before_update() 115
- before_validation() 115
- before_validation_on_create() 115
- before_validation_on_update() 115
- belongs_to 29
- belongs_to() 77
 - :class_name 83
 - :conditions 83
 - :counter_cache 83, 88
 - :foreign_key 83
 - :order 83, 88
 - :polymorphic 96
 - Unterschied zu has_one() 83
- button_to() 132, 176
- cache() 190
- cache_sweeper() 190
- caches_action() 188
- caches_page() 188
- Caching 187
 - ActionController::Base.fragment_
 - cache_store 192
 - ActionController::Caching::Fragments 192
 - cache() 190
 - cache_sweeper() 190
 - caches_action() 188
 - caches_page() 188
 - expire_action() 189
 - expire_fragment() 191
 - expire_page() 189
 - read_fragment 190
 - Statischer Seitenbereich 190
 - Sweeper 189
 - Testen 193
- Callbacks 115
 - Überschreiben 116
- Callback-Makros 116
- Callback-Objekte 117
- Capfile 312
- Capistrano 309
 - :application 330
 - :deploy_to 314
 - :repository 313
 - :svn 318
 - :use_sudo 315
 - :user 314
- After-Tasks 325
- Before-Tasks 325
- cap:deploy 319
- cap:deploy:cold 316
- Capfile 312
- Datenbankmigration 319
- delete() 324
- deploy.rb 312
- deploy:migrate 319
- deploy:migrations 320
- deploy:rollback 320
- Fallstricke 318
- FastCGI-Prozesse starten 315
- Installation 312
- Konfiguration 313
- LightTPD konfigurieren 316
- LightTPD starten 316
- on_rollback 324
- put() 324
- Quickstart 310
- Reaper 328
- Referenz 330
- remote:exec 321
- remote:setup 315
- render() 324
- Rollback 320
- Rollen 314
- run() 321, 323
- Setup 315
- Shared Verzeichnis 328
- spawn-fcgi 318
- Spawner 315, 327
- Spinner-Task 315

- Standardtasks überschreiben 325
- Subversion 311
- sudo() 321, 323
- Tasks 320
- Tasks ausführen 321
- Tasks und Rollen 322
- Transaktionen 324
- Variablen 326
- capture() 178
- check_box() 169
- Cluster 294
- collection_select 55
- column_names() 75
- columns() 75
- composed_of() 99
- connect() 147
- consider_all_requests_local 293
- console 63
- Console-Skript 306
- Constraint 24
- content.columns() 75
- content_for() 178
- content_tag() 171
- Controller 121
 - REST 199
 - testen 352
- Controller-Test
 - get() 352
 - HTTP-Requesttypen 352
- controls.js 265
- Cookie als Session-Speicher 138, 300
- Cookie für Session-ID 138
- create() 65
- CRUD 26, 64
- curl 134
- date_select() 172
- Datei-Download 156
- Datei-Upload 158
- Datenbank als Session-Speicher 300
- Datenbank-Constraint 24
- Datenbankkonfiguration 18
- Datenbankmigration 392
 - add_column() 393
 - add_index() 394
 - Capistrano 319
 - change_column() 394
 - change_column_default() 394
 - create_table(name, options) 392
 - drop_table(name) 393
 - Generator 25, 392
 - Migrationsskript ausführen 392
 - Migrationsskript erzeugen 392
 - remove_column() 393
 - remove_index() 394
 - rename_column() 393
 - rename_table() 393
 - SQL ausführen 394
- Debugging
 - Ausgabe von Objekten im View 179
- delete() 68
- delete_all() 68
- deploy.rb 312, 331
- Deployment 291
 - Capistrano 309
- destroy() 68
- destroy_all() 68
- Direkteingabe einer URL mit ID 135
- DOM 264
- Domain-Modell 16
- DoubleRenderError 126, 131
- Download von Dateien 156
- Drag and Drop 275
 - acts_as_list() 280
 - Beispiel 275
 - has_many() 280
 - Sortieren 278
- dragdrop.js 265
- draggable_element() 275
 - :revert 275
- DRY 8
- DSL 385
- Dynamische Finder 73
- E-Mail, Versand bei Ausnahme 306
- effects.js 265
- Entwicklungsumgebung 292
- ERb 127
- errors() 110
 - on() 110
- error_message_on() 185
- error_messages_for() 185

- establish_connection() 63
- Exception_Notifier 306
- expire_action() 189
- expire_fragment() 191
- expire_page() 189

- Feedback 16
- fields_for() 167
- Filter 141
 - Ausführungskette 142
 - Bedingungen 143
 - before_filter 51
 - Filterketten 145
 - Vererbung 144
- filter() 144
- Filterketten 145
- Filterklassen 144
- find() 65
 - :all 66
 - :conditions 68
 - :first 66
 - :include 72
 - :joins 72
 - :limit 71
 - :offset 71
 - :order 71
 - asc 71
 - desc 71
- find_all_by() 73
- find_by() 73
- find_by_sql() 74
- Fixtures 350, 364
 - Verzeichnis 364
 - Verzeichnisse 346
- fixtures() 365
- Flash Speicher 140
 - Redirects 140
- flash() 140
 - keep() 141
 - now() 141
- form_for() 165
- form_remote_tag() 267, 269
 - :position 270
 - :update 267
 - :url 267
- form_tag 31
- FormHelper 169
- Formular 168
- Fragment-Caching 190
- Fremdschlüssel 28, 76
- Funktionale Tests 373

- Generator 20
- GET-Request 131
- Gettext 231
- Globalize-rails 227

- has_and_belongs_to_many()
 - :association_foreign_key 93
 - :class_name 92
 - :conditions 93
 - :delete_sql 93
 - :finder_sql 93
 - :foreign_key 92
 - :insert_sql 94
 - :join_table 92
 - :order 93
 - :uniq 93
 - erzeugte Methoden 91
- has_many 33
- has_many() 76, 85
 - :as 96
 - :counter_sql 87
 - :dependent 87
 - :finder_sql 87
 - :through 97
- has_one() 78
 - :class_name 79
 - :conditions 80
 - :dependent 81
 - :foreign_key 81
 - :order 80
 - Unterschied zu belongs_to() 83
- Helper-Modul 163
- hidden_field() 170
- HTML-Formular 168
- HTML-Template 161
- HTTP
 - Accept 133
- HTTP-Authentication 146
- HTTP-Code 404 293
- HTTP-Code 422 293
- HTTP-Code 500 293

human_size() 172

Iconv 230

image_tag() 178

Inline-Filter 144

InnoDB 103, 367

Installation

 Capistrano 312

 Instant Rails 12

Instant Rails 12

Instanzvariablen

 im Layout 180

 im View 163

Integrationstests 383

 Sessions 386

Internationalisierung 227

 \$KCODE 231

 Anwendung vorbereiten 230

 Bibliothek jcode 231

 Checkliste 229

 Datenbank und Zeichensatz 230

Gettext

 Controller 238

 Datum, Zahlen, Währungen 243

 Datumsfelder 245

 Dynamische Texte 241

 Fehlermeldungen 239

 init_gettext() 237

 Lokalisierte Templates 238

 Methode _() 233

 Methode gettext() 233

 mo-Datei 232

 Modelle übersetzen 239

 Monate 245

 Pluralisierung 242

 po-Datei 232

 pot-Datei 232

 Rake-Task 233

 Sortierung 247

 Wochentage 245

GNU Gettext 232

HTML und Zeichensatz 231

Iconv 230

Ruby-Gettext-Package 231

UTF-8 230

InvalidAuthenticityToken 167

javascript_include_tag() 177

keep() 141

Konfiguration

 Capistrano 313

Konvention über Konfiguration 7

Layout 44, 153, 180

 Action-spezifisch 155

 dynamische Bestimmung 155

 yield 153

layout() 154

 :except 155

Layoutzuweisung

 automatisch 154

 Vererbung 154

 explizit 154

 Vererbung 154

Lazy Loading 73

LightTPD konfigurieren 316

LightTPD starten 316

link_to() 132, 175

link_to_remote() 266

 :update 266

Listen 105

local_request?() 293

lock_version 119

Locking 119

Log-Dateien 305

Lokalisierung 228

mail_to() 177

Massenzuweisung schützen 129

Metadaten 75

Migration 392

 Siehe Datenbankmigration 22

 timestamps 24

Migrationsskript 22

Mock-Objekte 387

Modellattribute testen 349

Modelllogik testen 351

Mongrel

 Anzahl Instanzen 301

 und Swiftply 301

Monitoring 306

MVC-Architektur 7

- N:M-Beziehungen 90
- Named Routes 150
- new 65
- Nginx und Mongrel 298
- now() 141
- number_with_delimiter() 171
- observe() 118
- observe_field() 270
 - :frequency 271
 - :update 271
 - :url 271
 - :with 271
- Tasks filtern 272
- Observer 118
- Optimistisches Locking 119
- options_for_select() 174
- options_from_collection_for_select() 175
- params 127
- Partials View 181
 - Fremder Controller 185
 - Instanzvariable 183
 - Layouts 184
 - Lokale Variable 182
 - Parameter 182
- Passwörter in Produktion 303
- password_field_tag() 170
- perform_caching 187, 293
- Performanz 301
- Plugin
 - Exception_Notifier 306
- Polymorphe Assoziationen 94
- POST-Request 131
- prepend_after_filter() 145
- prepend_before_filter() 145
- Pretty URLs 147
- Primärschlüssel 60
- Product Backlog 17
- Produktion
 - 404.html 293
 - 422.html 293
 - 500.html 293
 - Alternative Webserver 298
 - Anwendung 291
 - Anwendung warten 305
 - Anzahl Mongrel-Instanzen 301
 - Apache und Mongrel 295
 - Backup 306
 - Cluster aus Webservern 294
 - consider_all_requests_local 293
 - Console-Skript 306
 - Cookie als Session-Speicher 300
 - Datenbank als Session-Speicher 300
 - E-Mail, Versand bei Ausnahme 306
 - Entwicklungsumgebung 292
 - Exception_Notifier 306
 - local_request?() 293
 - Log-Dateien 305
 - Monitoring 306
 - Nginx und Mongrel 298
 - Passwörter 303
 - perform_caching 293
 - Performanz 301
 - Produktionsumgebung 292
 - RAILS_ENV 291
 - Session-Speicher 300
 - Sessions aufräumen 305
 - Sicherheit 302
 - Swiftply 301
 - Testumgebung 292
 - Überwachung 306
 - Umgebung definieren 291
 - Umgebung hinzufügen 293
 - Wartung 305
 - WEBrick 295
- Produktionsumgebung 292
- protect_from_forgery() 167
- Prototype 265
- prototype.js 265
- Quickstart
 - Capistrano 310
- radio.button 169
- radio.button_tag() 170
- Rails-Konsole für Controller 158
- Rails-Umgebungen 291
- RAILS_ENV 291
- Rake 347
 - clone_structure_to_test 347
 - recent 347
 - test:functionals 347
 - test:units 347

- rakefile 347
- raw_post() 271
- read_fragment 190
- Reaper 328
- redirect.to() 131
 - :controller 131
- Redirects 130
 - Reload 130
- Refactoring 355
- Reflection 122
- reload() 66
- render() 124
 - :action 124
 - :layout 125, 156
 - :nothing 125
 - :status 125
 - :template 124
 - :text 125
 - xml 134
- Request Parameter 127
- rescue_from 136
- reset_session() 139
- respond_to() 133
- Responses 123
- REST 195
 - ActiveResource 225
 - AJAX 222
 - Controller 199
 - Eigene Actions 219
 - Eigene Respond-Formate 221
 - Formatangabe via URL 203
 - Grundlagen 196
 - HTTP-Header 202
 - HTTP-Methoden 196
 - Mime-Types 222
 - Path-Methoden 204
 - respond_to 201
 - Ressource 196
 - Ressource Scaffolding 198
 - REST URLs in Views 204
 - Routing 210
 - Anpassung 210
 - map 210
 - name_prefix 210
 - path_prefix 211
 - singular 210
 - Unit-Tests 224
 - URL-Methoden im Controller 209
 - URLs 200
 - Verschachtelte Ressourcen 211
 - Vorteile 197
- Rollback
 - Capistrano 320
- Rollen
 - Capistrano 314
- Routes
 - Named 150
 - Root 150
- routes.rb 147
- Routing 147
 - Konfiguration 149
 - link.to() 153
 - map.connect() 147
 - redirect.to() 153
 - REST 210
 - routes.rb 147
- Ruby-Gettext-Package 231
- save() 66
- save_with_validation() 110
- Scaffolding 353
- schema.info 24
- script/console 63
- Scriptaculous 265
- select() 174
- select_tag() 171
- send_data() 156
 - :disposition 156
 - :filename 156
 - :type 156
 - data 156
- send_file() 157
 - :buffer_size 157
 - :disposition 157
 - :stream 157
 - :type 157
 - filename 157
 - sichere Verwendung 157
- Session 52
 - Daten löschen 139
- session() 138
- Session-Erzeugung abschalten 139

- Session-ID über Cookie 138
- Session-Speicher
 - Cookie 138
 - in Produktion 300
- Sessions 137
- Sessions aufräumen 305
- set_primary_key() 60
- set_table_name() 61
- setup() 338, 362
- Sicherheit
 - HTML-Code filtern 179
- Sicherheit in Produktion 302
- Single Table Inheritance 101
- Skalierung 299
 - horizontal 299
- sortable_element()
 - :update 278
 - :url 278
- Spawner 327
- SQL verwenden 74
- SQL-Injection vermeiden 70
- strip_links() 172
- stylesheet_link_tag() 177
- Subversion 311
- Swiftiply 301
- Tabellennamen 61
- tag() 171
- Tasks
 - Capistrano 320
- TDD 333
- TDD-Kreislauf 341
- teardown() 338, 362
- Template, Action Mailer 250
- Template, HTML 161
- Template, XML 186
- Test
 - Ausführen 348
- Test-DSL 385
- Test::Unit 336
 - Assertion 337
 - Test 337
 - Test Case 337
 - Test Suites 338
- Test::Unit::TestCase
 - use_transactional_fixtures 367

- Testdatenbank 346
 - Erstellung 346
 - Konfiguration 346
- Testen Caching 193
- Testgenerierung 345
- Testgetriebene Softwareentwicklung 333, 340
- Testumgebung 292
- Testverzeichnisse 346
- text_area() 169
- text_field() 129, 168
- text_field_tag() 170
- text_field_with_auto_complete() 281
- Transaktionale Fixtures 367
- Transaktionen 103
 - Capistrano 324
 - InnoDB 103
 - save
 - () 104
- truncate() 172
- Umgebung definieren 291
- Umgebung hinzufügen 293
- Umgebungen in Rails 291
- Unit Tests 333
 - assert_generates() 376
 - assert_no_tag() 379
 - assert_raise() 372
 - assert_recognizes() 376
 - assert_redirected_to() 375
 - assert_response() 374
 - assert_routing() 377
 - assert_tag() 377, 382
 - assert_template() 379
 - Assertions – Übersicht 389
 - assigns() 377
 - Ausführen 339
 - Beispiel 334
 - Benennung von Testklassen 338
 - Controller-Tests 373
 - cookies() 377
 - CRUD-Test 367
 - Datencontainer-Assertions 377
 - Definition 333
 - delete() 373
 - Dynamische Sprachen 336

- E-Mails testen 387
- Fixtures 364
- flash 380
- flash() 377
- get() 373
- head() 373
- Initialisierung von Template-Objekten 382
- Instanzvariablen 381
- Integrationstests 383
- Kontrollfluss-Assertions 374
- Lange Ausführungszeiten 365
- Mock-Objekte 387
- Modellklassen 362
 - Was testen? 363
- post() 373
- put() 373
- Rake 347
- Routing-Assertions 376
- session 379
- session() 377
- setup() 338
- TDD-Kreislauf 341
- teardown() 338
- Template-Assertions 377
- Test von Relationen 369
- Testrezepte
 - für Controller und Views 379
 - für Modelle 367
- Tests für Fehlerbehandlung 372
- TODO-Listen 342
- Unabhängigkeit 340
- Validierung 370
- Views testen 381
- Warum Testen? 335
- Werte im View 382
- update() 67
- update_all() 67
- Upload von Dateien 158
- URL, Direkteingabe mit ID 135
- URL-Generierung 152
- url_for() 152
- validate() 110
- validate_on_create() 111
- validate_on_update() 111
- validates_acceptance_of() 112
- validates_associated() 112
- validates_confirmation_of() 112
- validates_each() 113
- validates_exclusion_of() 113
- validates_format_of() 113
- validates_inclusion_of() 113
- validates_length_of() 114
- validates_numericality_of() 115
- validates_presence_of 46
- validates_presence_of() 111
 - :message 111
 - :on 111
- validates_uniqueness_of() 115
- Validierung 45, 110
- Validierungs-Klassenmethoden 111
- Validierungsfehler, Anzeige 185
- Variablen
 - Capistrano 326
- Vererbung 100
 - Base.inheritance_column() 102
 - type 102
- verify() 133
- verify_authenticity_token 167
- Versteckte Felder 170
- View 161
- Views testen 353
- Wartung der Anwendung 305
- WEBrick 295
 - Start 19
- Webserver
 - Alternative 298
 - Apache und Mongrel 295
 - Nginx und Mongrel 298
 - WEBrick 295
- Webserver Cluster 294
- Willkommensseite 14
- XML-Template 186
- XMLHttpRequest 264
- YAML-Fixtures 364
- yield 180
- yield im Layout 153
- Zusammengesetzte Primärschlüssel 61