HANSER

Rapid Web Development mit Ruby on Rails

Ralf Wirdemann, Thomas Baustert

ISBN 3-446-40932-7

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/3-446-40932-7 sowie im Buchhandel

Kapitel 3

Hands-on Rails

In diesem Kapitel werden wir eine erste Web-Applikation mit Ruby on Rails entwickeln. Unser Ziel ist es, Ihnen alle wesentlichen Komponenten des Frameworks und den Rails-Entwicklungsprozess im Schnelldurchlauf zu präsentieren. Dabei gehen wir bei der Beschreibung 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 dieses Kapitels 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 Anmerkung 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 7.

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 zum Testen 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`. Sie `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

[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 14 und 15 gesondert zu behandeln, um dieses Kapitel nicht ausufern zu lassen.

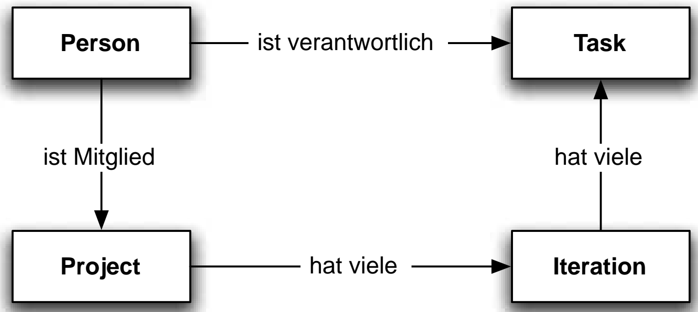

Abbildung 3.1: OnTrack-Domain-Modell

unseres 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 Backlog[2]. 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/apis
    create  app/controllers
    ...
$ 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 15). 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 15.2) und *production* für die Produktionsversion der Anwendung (siehe dazu Abschnitt 12.1).

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: mysql
  database: ontrack_development
  host: localhost
  username: rails
  password: secret
```

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

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

Wir verwenden für die OnTrack-Anwendung eine MySQL[4]-Datenbank. Wenn Sie eine andere Datenbank verwenden möchten, müssen Sie den Konfigurationseintrag unter `adapter` entsprechend ändern. Eine Liste von unterstützten Datenbanken finden Sie auf der Rails-Homepage[5]. Ebenso sind der Benutzername (wir verwenden für das Beipiel `rails`) und das Passwort einzutragen. Alle anderen Einstellungen sowie die Einstellungen der beiden Datenbanken *test* und *production* lassen wir zunächst unverändert.

Nachdem wir die Datenbank konfiguriert haben, müssen wir sie natürlich auch erstellen:

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

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*. Zum Starten des Servers geben wir den Befehl *ruby script/server webrick*[6] ein:

```
$ 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
```

3.5 Projekte erfassen, bearbeiten und löschen

Eine Projektmanagement-Software ohne eine 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.

[4] Siehe http://www.mysql.de
[5] http://wiki.rubyonrails.com/rails/pages/DatabaseDrivers
[6] Die Skripte einer Rails-Anwendung liegen im Verzeichnis *APP_ROOT/script*.

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

```
$ ruby script/generate model project
```

Der erste Parameter *model* gibt den Namen des Generators (hier für Modelle) und der zweite Parameter *project* den Namen der Modellklasse an. Die Ausgabe des Generators sieht in etwa wie folgt aus:

```
$ ruby script/generate model project
        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
```

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 14 und 15. Im Moment interessieren uns nur das Model 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 3.2: app/models/projects.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.

> ***Ruby: Klasse***
> Eine Klasse wird in Ruby durch das Schlüsselwort `class` eingeleitet. Die Vererbung wird durch <, gefolgt von der Superklasse, definiert. Im Listing 3.2 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

Seit Rails 1.0 werden Datenbankänderungen nicht (mehr) 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 3.3: db/migrate/001_create_projects.rb

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

Die Tabelle `projects` wird in unserem Beispiel in `up()` mit folgenden Attributen erzeugt und in `down()` wieder gelöscht:

Listing 3.4: db/migrate/001_create_projects.rb

```
class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      t.column :name, :string, :limit => 100,
               :null => false, :default => ""
      t.column :description, :text, :null => false, :default => ""
      t.column :start_date, :date, :null => false, :default => 0
    end
  end

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

Eine Attributdefinition enthält mindestens den Attributnamen (z.B. `:name`) und den Typ (z.B. `:string`). 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). Beachten Sie für den Fall, dass ein Attribut nicht `NULL` sein darf (`:null => false`), auch ein Defaultwert anzugeben ist (`:default => ...`). Andernfalls kommt es später bei der Ausführung des Skripts zu einem Problem. Eine Übersicht aller unterstützten Methoden, Datentypen und Parameter findet sich im Anhang unter dem Abschnitt 15.11.

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 den Schlüsseln `:limit`, `:null` und `:default` erzeugt und als Parameter an die Methode `column` 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]`.

Der Primärschlüssel (vgl. 4.1.2) mit Namen `id` wird von Rails automatisch in die Tabelle eingetragen. 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` (siehe Kasten rake):

```
$ rake migrate
(in .../ontrack)
== CreateProjects: migrating
-- create_table(:projects)
   -> 0.5113s
== CreateProjects: migrated (0.5115s)
```

Rails ermittelt hierbei aus der Tabelle `schema_info`[7] 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.

> ***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.

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 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.

[7] Diese 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 vom Generator übernommen wird:

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

Sie können ebenso auch 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 `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 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 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 erzeugen

Wie Modelle werden auch Controller initial über einen Generator erstellt. Dieser erzeugt eine leere Controllerklasse, die wir Schritt für Schritt um Methoden erweitern müssen:

```
$ ruby script/generate controller project
  ...
```

Im Beispiel verwenden wir aber alternativ den Scaffold[8]-Generator. Dieser erzeugt bereits (CRUD[9])-Quellcode zum Anlegen, Löschen und Bearbeiten von Modellen über den Controller. Der erzeugte Quellcode umfasst neben dem Controller selbst auch alle notwendigen HTML-Seiten zur Anzeige und Bearbeitung des Modells. Der Aufruf sieht wie folgt aus:

```
$ ruby script/generate scaffold project project
  ...
```

Dem Scaffold-Generator werden als erster Parameter das Modell und als zweiter der Controller übergeben. Nur den Controller zu erzeugen, ist mit diesem Generator nicht möglich (siehe Kasten Scaffold-Generator). Aber keine Sorge, der Generator

[8] Zu Deutsch so viel wie Gerüstbau
[9] Das Akronym CRUD steht für *create*, *read*, *update* und *delete*.

überschreibt per Default keine bereits vorhandenen Dateien, sodass wir ihn nutzen können, obwohl wir das Modell und die zugehörigen Komponenten bereits angelegt haben. Werfen Sie nach dem Aufruf doch mal einen Blick in den erzeugten Quellcode. Vielleicht hilft er Ihnen als Vorlage bei Ihren ersten eigenen Schritten mit Rails.

> ***Rails: Scaffold-Generator***
> Der Scaffold-Generator erzeugt Quellcode für das Modell, den Controller, mehrere HTML-Seiten usw. Der Quellcode für den Controller und die HTML-Seiten enthalten dabei Zugriffe auf die Attribute der erzeugten Modellklasse. Da die Modellklasse ihre Attribute über die zugehörige Datenbanktabelle ermittelt, muss die Tabelle vor der Generierung des Quellcodes existieren. Ist die Tabelle beim Aufruf des Generators nicht vorhanden, erhalten wir die Meldung *error Before updating scaffolding from new DB schema, try creating a table for your model*. Daher ist bei der Nutzung des Scaffold-Generators zuerst das Modell mit Hilfe des Modellgenerators zu erstellen. Über das hierbei erzeugte Migrationsskript wird danach die Datenbanktabelle angelegt. Im Anschluss kann der Scaffold-Generator für die Generierung des Controllers und der Views verwendet werden. Per Default werden existierende Dateien nicht überschrieben, sodass das bereits erstellte Modell erhalten bleibt. Durch den Aufruf *ruby script/generator scaffold* wird die Beschreibung ausgegeben.

Damit sind die Entwicklungsarbeiten für unseren ersten Backlog-Item bereits abgeschlossen, und die Anwendung kann gestartet werden. Öffnen Sie die Startseite *http://localhost:3000/project* der Anwendung in einem Browser und prüfen Sie es selbst. Schon jetzt stehen Ihnen die Seiten aus Abbildung 3.2 zur Verfügung. Groovy, äh ruby oder?

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 compiliert 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 und Migrationsskripts per Modellgenerator: *ruby script/-generator model MODEL.*
2. Definition der Datenbanktabelle etc. im Migrationsskript und Datenbankaktualisierung per *rake migrate*.

3. Erzeugung des initialen Controllers per *ruby script/generator controller CONTROLLER* und Erstellen der HTML-Seiten etc. von Hand oder Nutzung des Scaffold-Generators: *ruby script/generate scaffold MODEL CONTROLLER*.
4. Aufruf *http://localhost:3000/CONTROLLER* und sich freuen.

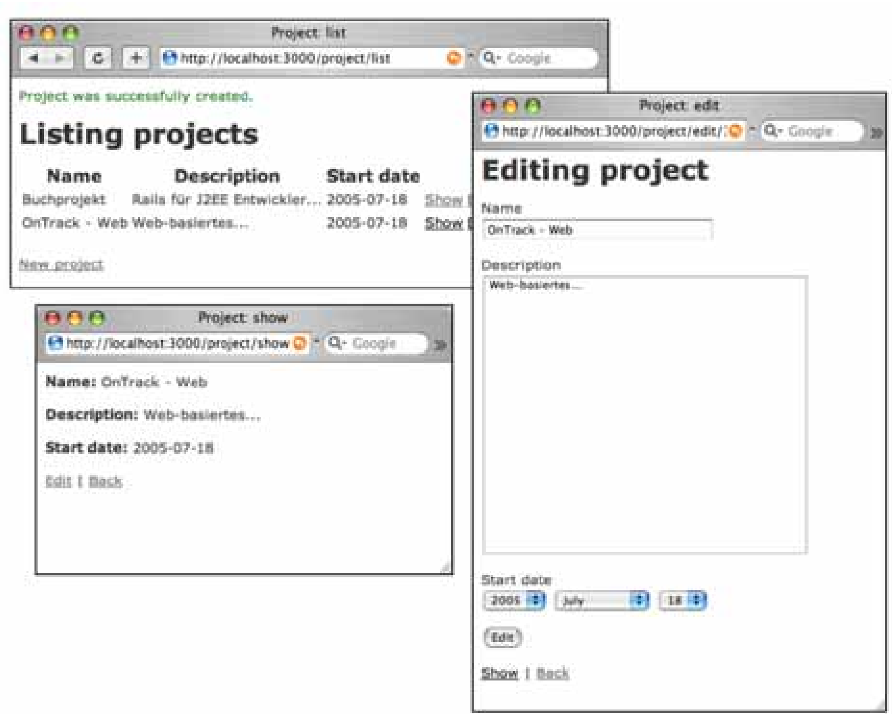

Abbildung 3.2: OnTrack in Version 0.1

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
  exists  app/models/
  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
```

Als Nächstes definieren wir wieder die Datenbanktabelle im Migrationsskript:

Listing 3.6: db/migrate/002_create_iterations.rb

```
class CreateIterations < ActiveRecord::Migration
  def self.up
    create_table :iterations do |t|
      t.column :name, :string, :limit => 100,
               :null => false, :default => ""
      t.column :description, :text, :null => false, :default => ""
      t.column :start_date, :date, :null => false, :default => 0
      t.column :end_date, :date, :null => false, :default => 0
      t.column :project_id, :integer, :null => false, :default => 0
    end
  end

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

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

```
$ rake migrate
(in .../ontrack)
== CreateIterations: migrating
-- create_table(:iterations)
   -> 0.0299s
== CreateIterations: migrated (0.0301s)
```

Das Feld *project_id* der Tabelle *projects* aus Listing 3.6 modelliert die N:1-Beziehung auf Datenbankebene, d.h. *project_id* ist ein Fremdschlüssel, der 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 3.7 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 Klas-

senmethode `belongs_to()` erweitert (siehe Kasten Klassenmethode). Ihr wird der Name des assoziierten Modells übergeben:

Listing 3.7: 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).

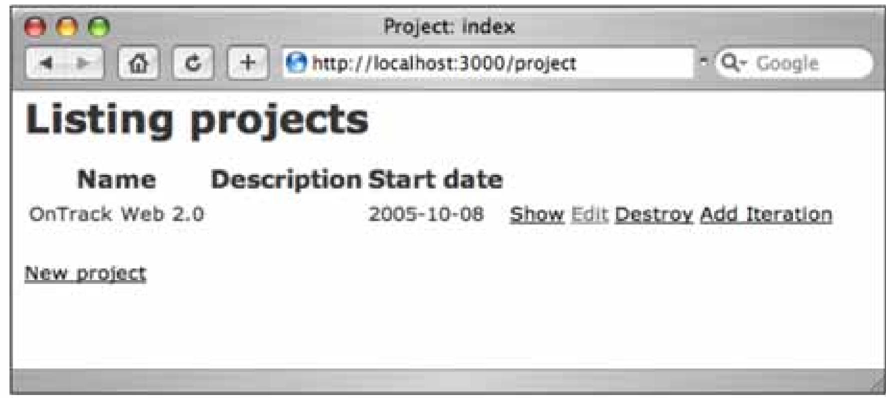

Abbildung 3.3: Ein neuer Link zum Hinzufügen von Iterationen

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 List-View um einen Aufruf des Formular-Helpers `link_to()`:

Listing 3.8: app/views/project/list.rhtml

```
...
<td><%= link_to 'Destroy', {
          :action => 'destroy',
          :id => project},
          :confirm => 'Are you sure?' %></td>
<td><%= link_to 'Add Iteration',
          :action => 'add_iteration',
          :id => project %></td>
...
```

Formular-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 Formular-Helper erfahren Sie in den Abschnitten 6.2 und 6.3.

Der Formular-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, die verschiedene Angaben enthält. Die Hash enthält zwei Schlüssel: `:action` und `:id`.

Über `:action` wird die Controller-Action definiert, 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 Projekt-ID an, die der Action `add_iteration()` beim Hinzufügen einer neuen Iteration übergeben wird.

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

Listing 3.9: app/controllers/project_controller.rb

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

Ruby: Methodendefinition
Eine Methode wird durch das Schlüsselwort `def` eingeleitet und mit `end` beendet. Bei keinem oder einem Parameter können die Klammern 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). Der Zugriff ist auch über die Instanzvariable `@params` möglich (siehe Kasten Instanzvariable), sollte aber vermieden werden. Die Methode versteckt die Implementierungsdetails und macht hier spätere Änderungen einfacher (siehe auch *http://weblog.rubyonrails.org/2006/4/25/use-params-not-params*).

In unserem Beispiel übergibt der neue Link *add_iteration* der Action den Parameter `:id`, dessen Wert die ID des Projekts angibt, dem die neue Iteration hinzugefügt werden soll. 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 *app/views/iteration/edit.rhtml* den View, der als Ergebnis zurückgeliefert wird. Dazu ist als Nächstes das Template zu erstellen:

Listing 3.10: app/views/iteration/edit.rhtml

```
<h1>Editing Iteration</h1>
<%= start_form_tag :action => 'update_iteration',
                   :id => @iteration %>
  <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_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

Der Formular-Helper `start_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 `ProjectController` 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 Übertragung der Projekt-ID an den Server benötigt wird. Der neue View ist in Abbildung 3.4 dargestellt.

Der Edit-View gibt als Submit-Action `update_iteration()` an, die wir im `ProjectController` implementieren:

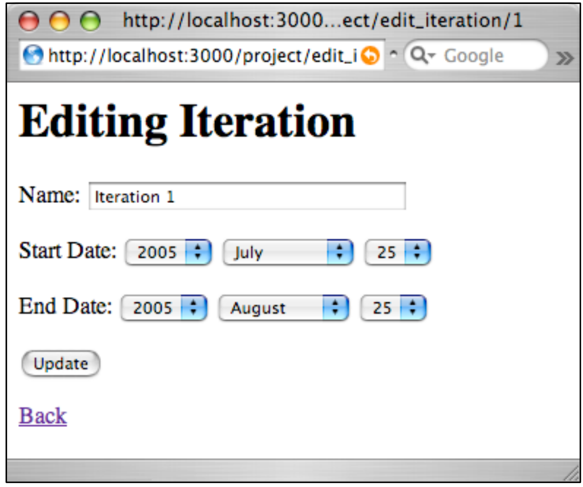

Abbildung 3.4: Ein View zum Bearbeiten von Iterationen

Listing 3.11: app/controllers/project_controller.rb

```
def update_iteration
  @iteration = Iteration.new(params[:iteration])
  if @iteration.save
    flash[:notice] = 'Iteration was successfully created.'
    redirect_to :action => 'list'
  else
    render :template => 'iteration/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-Bedingung). Wenn ja, dann erfolgt eine Weiterleitung auf die Action `list()`, welche eine Liste aller Projekte anzeigt. 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-Bedingung***
> 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 3.12: 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. Der folgende Code-Auszug zeigt die Verwendung dieser Methode im List-View für Projekte:

Listing 3.13: app/views/project/list.rhtml

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

  <% for project in @projects %>
    <tr>
    <% for column in Project.content_columns %>
      <td><%=h project.send(column.name) %></td>
    <% end %>
    <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 wird über die Klassenmethode `content_columns()` ein Array aller Spalten der Datenbanktabelle `projects` ermittelt, die für eine Anzeige sinnvoll sind (z.B. wird die Spalte `id` weggelassen). Für jede Spalte wird ihr Wert ausgegeben. Dabei ist die Methode `h()` ein Alias für `html_escape()` und konvertiert HTML-Elemente, z.B. < in `<` (siehe Abschnitt 6.1). Die Methode `Project.send()` ist hier nur eine andere Schreibweise für `project.column.name`.

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 3.5).

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 kennen gelernt und um einen zusätzlichen Link erweitert. Die von dem Link referenzierte Action `add_iteration()` haben wir als neue öffentliche Methode der Klasse `ProjectController` 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()` kennen gelernt.

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 3.14: app/views/project/show.rhtml

```
...
<h2>Iterations</h2>
<table>
<tr>
  <th>Name</th>
  <th>Start</th>
  <th>End</th>
</tr>
<% for iteration in @project.iterations %>
```

```
<tr>
  <td><%= 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 3.15: app/controllers/project_controller.rb

```
def show_iteration
  @iteration = Iteration.find(params[:id])
  render :template => 'iteration/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 3.16: app/views/iteration/show.rhtml

```
<% 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 3.17: app/views/project/show.rhtml

```
...
<% 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 `ProjectController` wie folgt implementiert wird:

Listing 3.18: app/controllers/project_controller.rb

```
def edit_iteration
  @iteration = Iteration.find(params[:id])
  render :template => 'iteration/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/iteration/edit.rhtml*.

Allerdings haben wir jetzt ein kleines Problem: Der Edit-View für Iterationen überträgt seine Daten an die von uns bereits implementierte Action `up-`

Abbildung 3.7: Neue Links: Edit und Destroy

`date_iteration()`. Da diese Action ursprünglich für die Neuanlage von Iterationen programmiert wurde, erzeugt und speichert die Action 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 3.19: app/views/iteration/edit.rhtml

```
<%= start_form_tag :action => 'update_iteration',
                   :id => @iteration %>
...
```

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 3.20: app/controllers/project_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 :action => 'show',
                :id => @iteration.project_id
```

```
    else
      render :template => 'iteration/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()`).

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

Listing 3.21: app/controllers/project_controller.rb

```
def destroy_iteration
  iteration = Iteration.find(params[:id])
  project = iteration.project
  iteration.destroy
  redirect_to :action => 'show', :id => 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
  ...
```

Als Nächstes definieren wir die Tabelle *tasks* im Migrationsskript:

Listing 3.22: db/migrate/003_create_tasks.rb

```
class CreateTasks < ActiveRecord::Migration
  def self.up
    create_table :tasks do |t|
      t.column :iteration_id, :integer,
               :null => false, :default => 0
      t.column :name, :string, :limit => 200,
```

```
                :null => false, :default => ""
       t.column :priority, :integer,
                :null => false, :default => 0
     end
   end

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

Im Anschluss aktualisieren wir die Datenbank wieder per:

```
$ rake migrate
```

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

Listing 3.23: 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 3.24: app/views/project/show.rhtml

```
...
<% for iteration in @project.iterations %>
<tr>
  <td><%= 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 zeigt den erweiterten View *project/show*.

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 `ProjectController`:

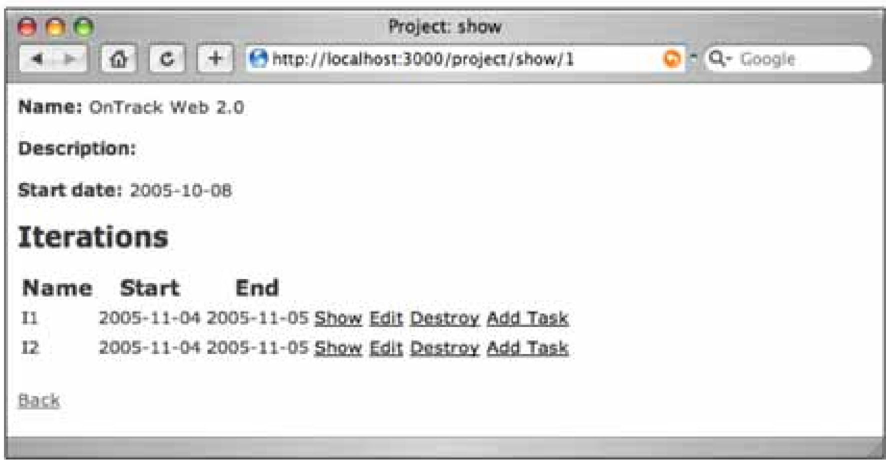

Abbildung 3.8: Der Show-View eines Projekts ermöglicht das Zufügen von Tasks

Listing 3.25: app/controllers/project_controller.rb

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

Das Vorgehen ist nahezu identisch wie das Hinzufügen von Iterationen zu Projekten. Die Action `add_task()` liefert den View *app/views/task/edit.rhtml*, den wir wie folgt implementieren:

Listing 3.26: app/views/task/edit.rhtml

```
<h1>Editing Task</h1>
<%= error_messages_for 'task' %>
<%= start_form_tag :action => 'update_task', :id => @task %>
<p><label for="task_name">Name:</label><br/>
  <%= text_field 'task', 'name'  %></p>
<p><label for="task_priority">Priority:</label><br/>
  <%= select(:task, :priority, [1, 2, 3]) %></p>
<p><label for="task_person_id">Responsibility:</label><br/>
  <%= collection_select(:task, :person_id,
  Person.find(:all, :order => "surname"), :id, :surname) %></p>
<%= submit_tag 'Update' %>
<%= hidden_field 'task', 'iteration_id' %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

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 3.27: app/controllers/project_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',
                :id => @task.iteration.project_id
  else
    render :template => 'task/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 3.28: app/views/project/show.rhtml

```
<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>
  ...
</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 3.29: 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 3.30: app/views/iteration/show.rhtml

```
<% 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><%= 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 *iteration/show*.

Der Show-Link verweist auf die Action `show_task()`, die wir im `ProjectController` implementieren:

Listing 3.31: app/controllers/project_controller.rb

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

Abbildung 3.9: Ein neuer Show-View für Iterationen

Der von der Action gelieferte *app/views/task/show.rhtml* 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 `ProjectController`-Action `edit_task()`:

Listing 3.32: app/controllers/project_controller.rb

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

Der zugehörige View *task/edit.rhtml* 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 `ProjectController` verlinkt:

Listing 3.33: app/controllers/project_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 die Instanzvariable `@content_for_layout` zur Verfügung. Genau an der Stelle im Layout, wo Sie diese Variable 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>
...
<%= @content_for_layout %>
...
</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 *project.rhtml*, die der Generator für unseren `ProjectController` 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. Es muss also wieder nichts programmiert werden.

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. Deshalb haben wir ein entsprechendes Logo erzeugt und in das Layout *app/views/layouts/project.rhtml* eingefügt:

Listing 3.34: app/views/layouts/project.rhtml

```
...
<div id="logo_b-simple">
  <img src="/images/logo.jpg" alt="b-simple" border="0">
</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 Stelle wieder zu verwenden. Wir werden darauf in Abschnitt 6.5 eingehen.

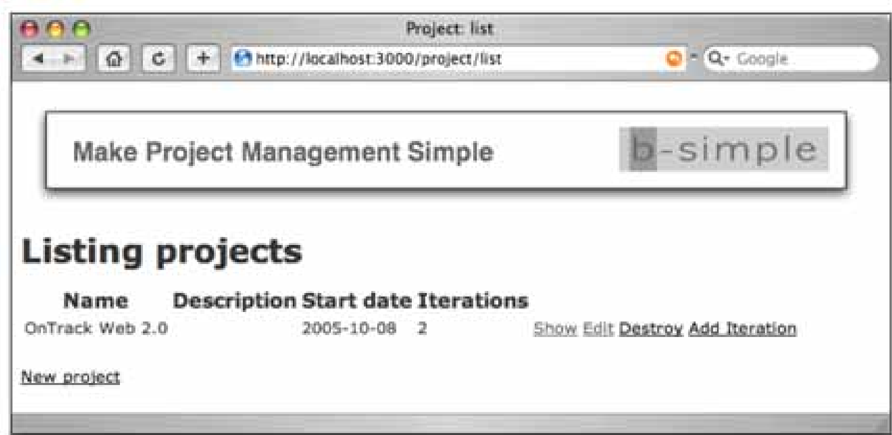

Abbildung 3.10: OnTrack-Seite mit Logo

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 4.13 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 3.35: 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 `ProjectController`-Action `new` aus, die den New-View eines Projekts wiederholt öffnet, wenn die Validierung fehlschlägt:

Listing 3.36: app/controllers/project_controller.rb

```
def create
  @project = Project.new(params[:project])
  if @project.save
    flash[:notice] = 'Project was successfully created.'
    redirect_to :action => 'list'
  else
    render :action => 'new'
  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) im Partial-View *views/project/_form.rhtml*[10] enthalten:

Listing 3.37: app/views/project/_form.rhtml

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

Der View verwendet den von Rails bereitgestellten 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.

Rails verwendet hier eine Standardfehlermeldung in Englisch. In Kapitel 7 zeigen wir Ihnen, wie Sie Ihre Anwendung internationalisieren bzw. lokalisieren und damit auch Fehlermeldungen z.B. in Deutsch anzeigen können.

[10] Scaffolding generiert hier einen Partial-View, der sowohl in *new.rhtml* als auch in *edit.rhtml* verwendet wird. Mehr über Partial-Views erfahren Sie in Abschnitt 6.5.

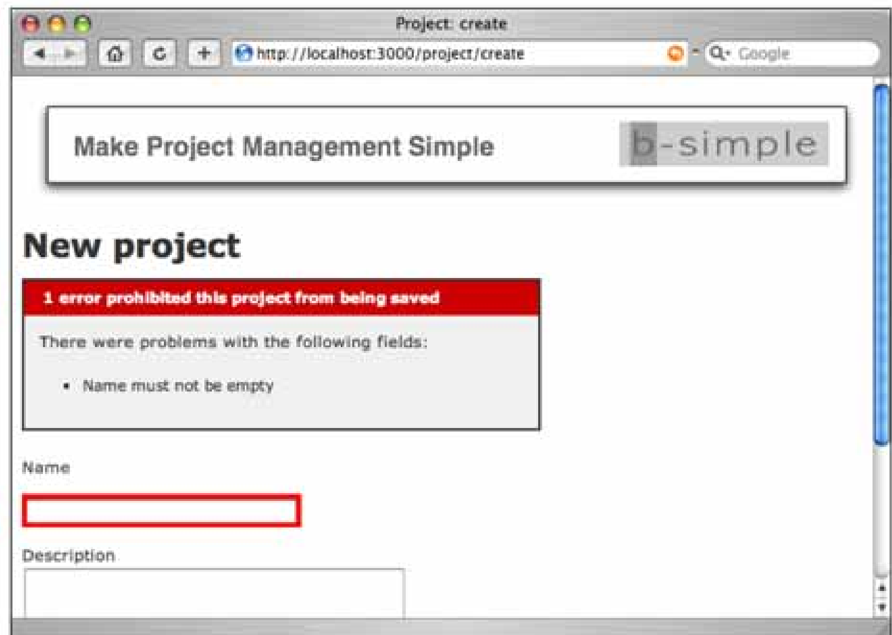

Abbildung 3.11: Projekte ohne Namen sind nicht erlaubt

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 3.38: 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).

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:

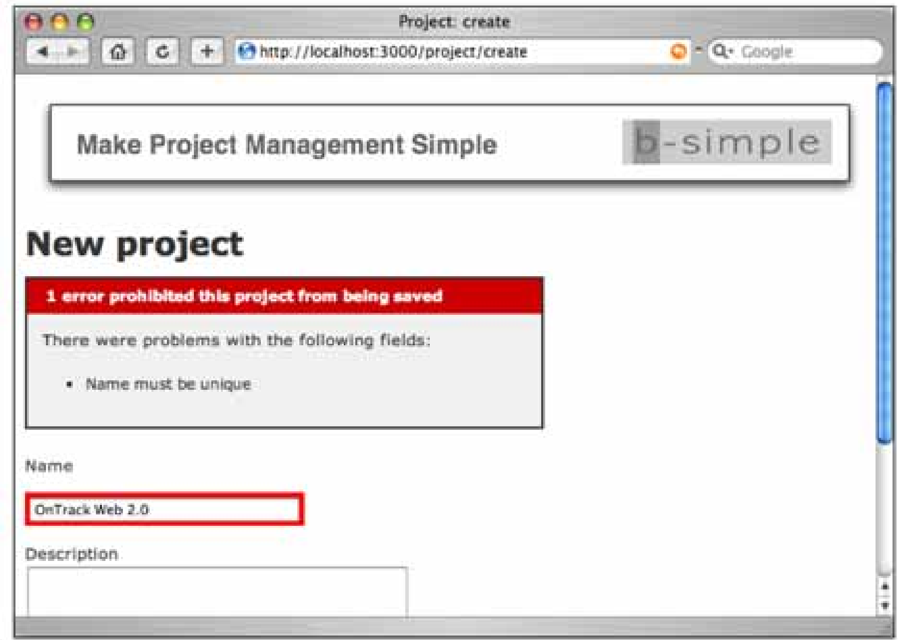

Abbildung 3.12: Projektnamen müssen eindeutig sein

Listing 3.39: app/models/iteration.rb

```
class Iteration < ActiveRecord::Base
...
protected
  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 3.20 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 (Sie werden es ahnen) ein Modell erzeugen:

```
$ ruby script/generate model person
```

Zum Speichern von Benutzern verwenden wir die Tabelle *people*:[11] Das folgende Skript enthält das entsprechende Schema und erzeugt zugleich einen Testbenutzer:

Listing 3.40: db/migrate/004_create_people.rb

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.column :username, :string, :limit => 50,
               :null => false, :default => ""
      t.column :password, :string, :limit => 50,
               :null => false, :default => ""
      t.column :firstname, :string, :limit => 50,
               :null => false, :default => ""
      t.column :surname, :string, :limit => 50,
               :null => false, :default => ""
    end
  end

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

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

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

```
$ rake migrate
```

Für die Verwaltung benötigen wir noch den Controller. Diesen nennen wir `AdminController`, da über ihn die Administration der Anwendung erfolgen soll.

```
$ ruby script/generate scaffold person admin
```

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

[11] Rails kennt einige spezielle Pluralisierungsregeln der englischen Sprache. Mehr zu diesem Thema finden Sie in Abschnitt 4.1.3.

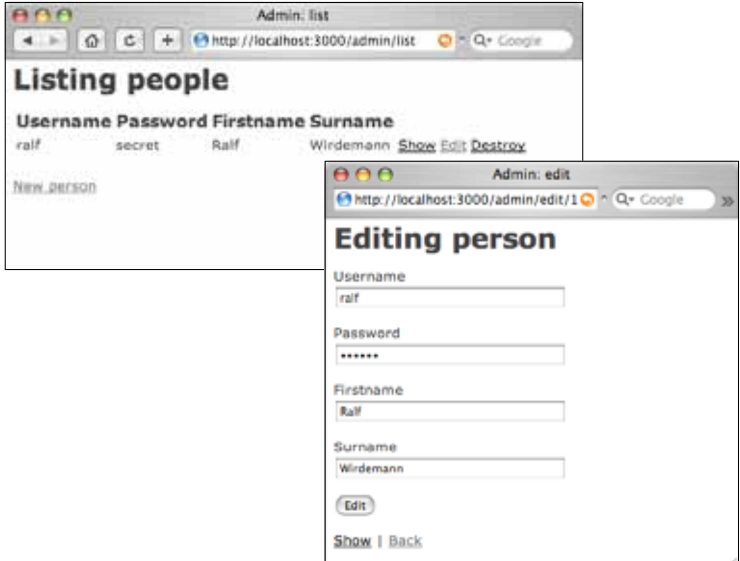

Abbildung 3.13: Die OnTrack-Benutzerverwaltung

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.[12]

Das erste für die Benutzerverwaltung benötigte neue Konzept, sind die so genannten *Filter* (vgl. Abschnitt 5.7). 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 am 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 3.41: app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  before_filter :authenticate, :except => [:login, :sign_on]

protected
```

[12] Für die Login-Funktionalität gibt es auch einen Generator, den wir hier aber nicht verwenden, weil wir die notwendigen Schritte explizit zeigen möchten.

```
  def authenticate
    redirect_to :controller => 'project', :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“).

Um zu testen, ob das Ganze funktioniert, haben wir in der ersten Version der Methode eine einfache Weiterleitung auf die `login()`-Action programmiert. Die eigentliche Prüfung haben wir erst mal weggelassen. Wir benötigen also zusätzlich eine neue Methode `login()` im `ProjectController`:

Listing 3.42: app/controllers/project_controller.rb

```
class ProjectController < ApplicationController
  def login
  end
  ...
end
```

Des Weiteren benötigen wir einen View *login*, der von der `login()`-Action aufgrund des gleichen Namens standardmäßig ausgeliefert wird. Dieser ist wie folgt zu implementieren:

Listing 3.43: app/views/project/login.rhtml

```
<%= form_tag :action => 'sign_on' %>
<table>
  <tr><td>Username:</td>
      <td><%= text_field 'person', 'username' %>
  </tr><tr>
    <td>Password:</td>
    <td><%= password_field 'person', 'password' %></td>
  </tr>
</table>
<%= submit_tag 'Login' %>
<%= end_form_tag %>
```

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

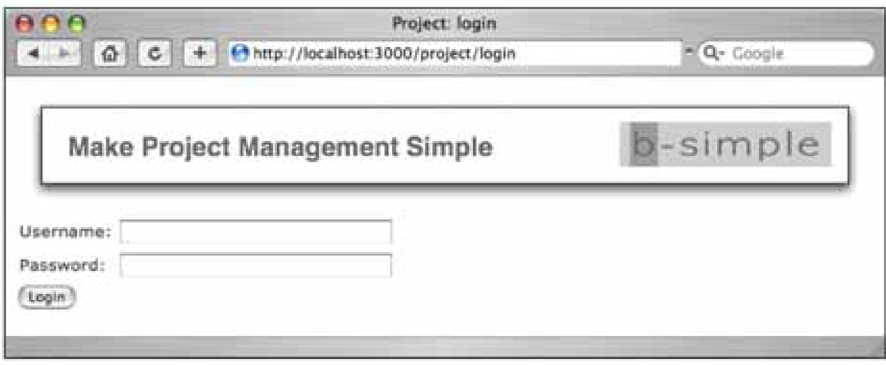

Abbildung 3.14: Die Login-Seite

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. Abschnitt 5.5). Wir nutzen die Session, um uns den erfolgreich angemeldeten Benutzer zu merken:

Listing 3.44: app/controllers/project_controller.rb

```
class ProjectController < ApplicationController
  def sign_on
    person = Person.find(:first, :conditions =>
      ["username = BINARY ? AND password = BINARY ?",
       params[:person][:username], params[:person][:password]])
    if person
      session[:person] = person
      redirect_to :action => 'list'
    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 3.45: app/controllers/application.rb

```
def authenticate
  unless session[:person]
    redirect_to :controller => 'project', :action => 'login'
  end
end
```

Wenn Sie wollen, können Sie das System noch um eine Abmelde-Action (und einen zugehörigen View) erweitern, bei der die Session per `reset_session()` zurückgesetzt wird.

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 alter_tasks
  exists  db/migrate
  create  db/migrate/005_alter_tasks.rb
```

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

Listing 3.46: db/migrate/005_alter_tasks.rb

```
class AlterTasks < ActiveRecord::Migration
  def self.up
    add_column :tasks, :person_id, :integer,
               :null => false, :default => 0
  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 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/task/edit.rhtml

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

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]`).

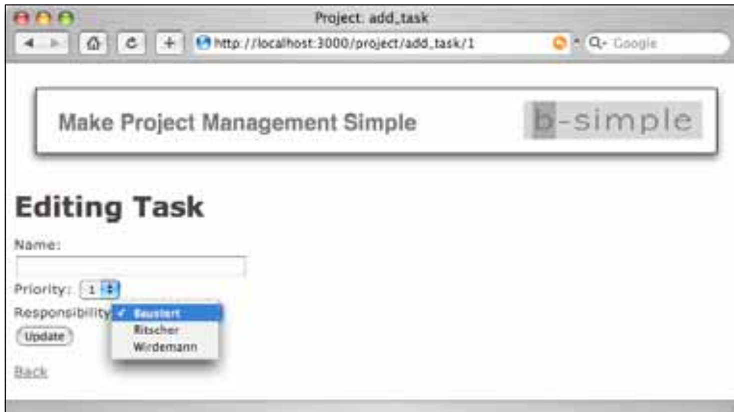

Abbildung 3.15: Zuordnung von Tasks

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 kennen gelernt:

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

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.

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