HANSER

Ralf Wirdemann, Thomas Baustert

Rapid Web Development mit Ruby on Rails

ISBN-10: 3-446-41498-3
ISBN-13: 978-3-446-41498-3

Leseprobe

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.

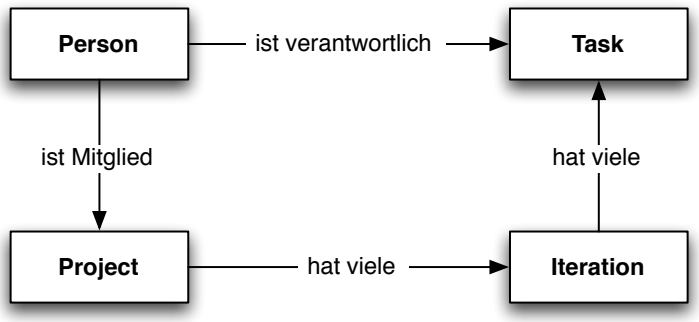

Abbildung 3.1: OnTrack-Domain-Modell

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 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/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-Homepage[5]. Wir verwenden für die OnTrack-Anwendung eine MySQL[6]-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 Mongrel[7] (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 Scaffold[9]-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
```

```
  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
```

```
      # 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.

[10] 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 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 CRUD[11]-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 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, 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/
```

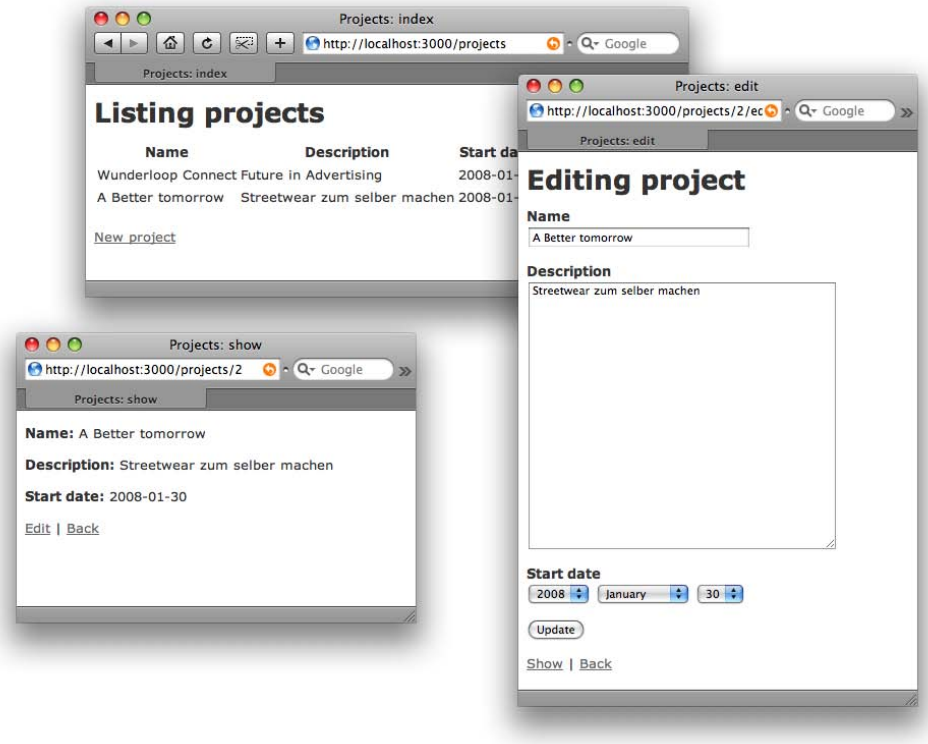

Abbildung 3.2: OnTrack in Version 0.1

```
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).

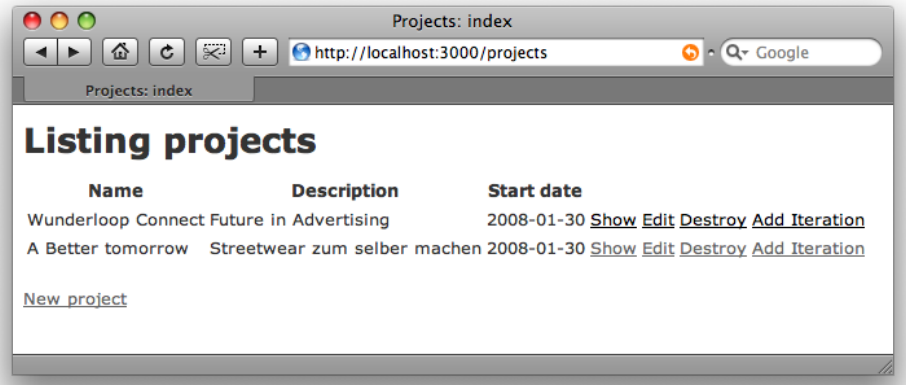

Abbildung 3.3: Ein neuer Link zum Hinzufügen von Iterationen

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.

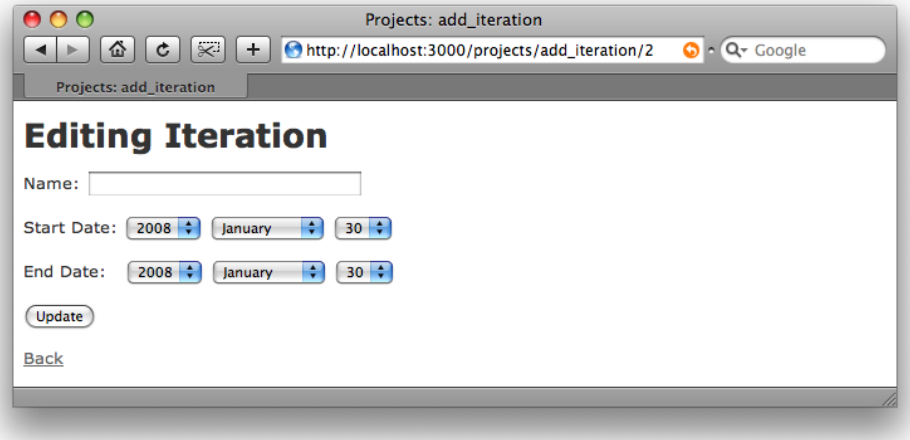

Abbildung 3.4: Ein View zum Bearbeiten von Iterationen

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).

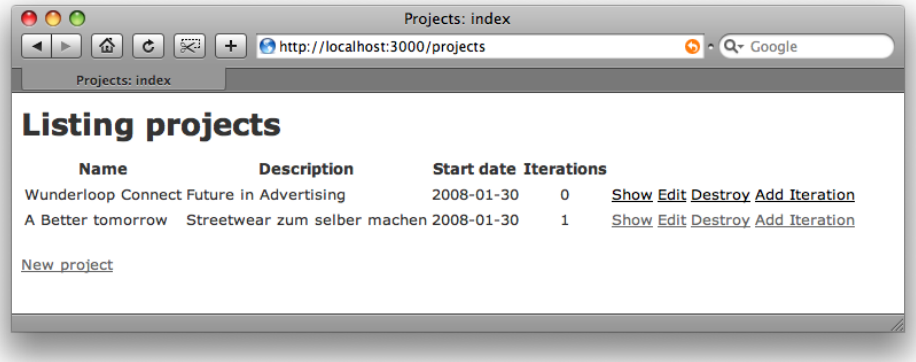

Abbildung 3.5: Projektübersicht inklusive Anzahl der Iterationen

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.

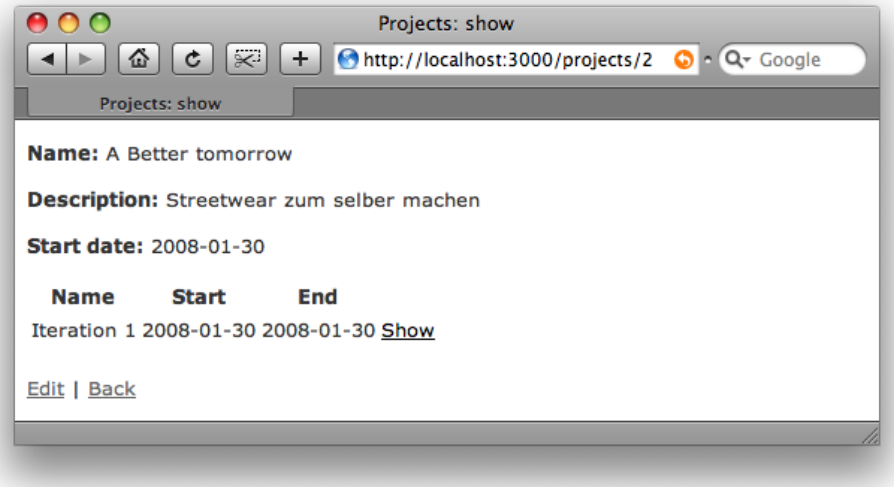

Abbildung 3.6: Ein Projekt mit Iterationen

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*.

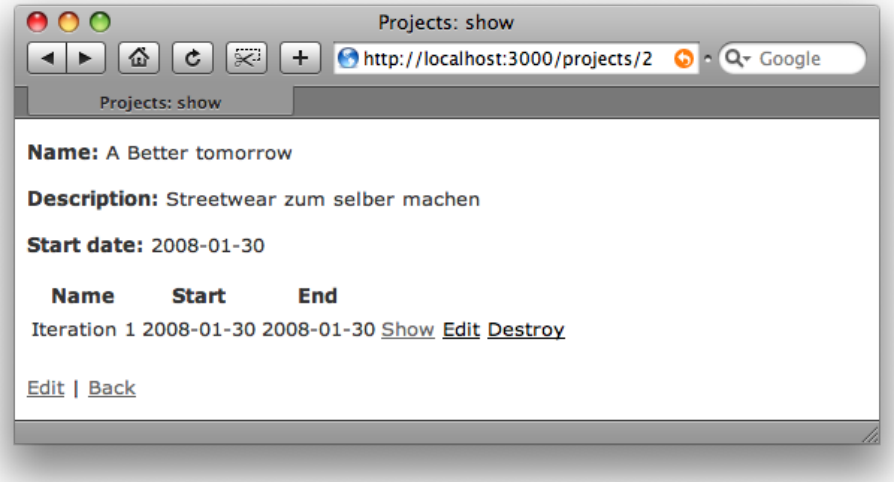

Abbildung 3.7: Neue Links: Edit und Destroy

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>
```

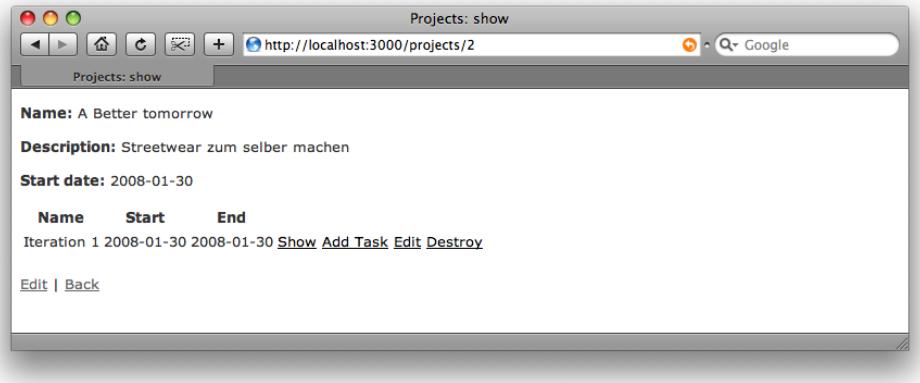

Abbildung 3.8: Der Show-View eines Projekts ermöglicht das Hinzufügen von Tasks

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>
  ...
</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
```

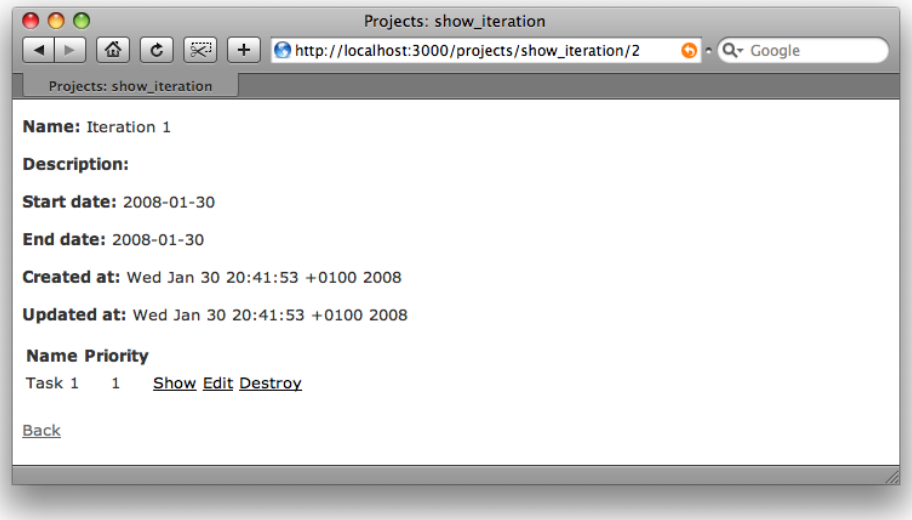

Abbildung 3.9: Ein neuer Show-View für Iterationen

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">
  <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 Stellen wieder zu verwenden. Wir werden darauf in Abschnitt 6.5 eingehen.

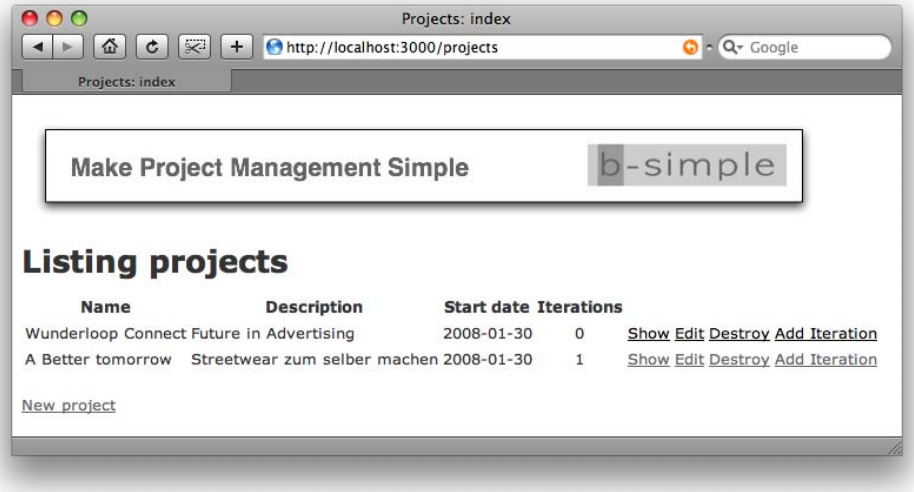

Abbildung 3.10: OnTrack-Seite mit Logo

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.

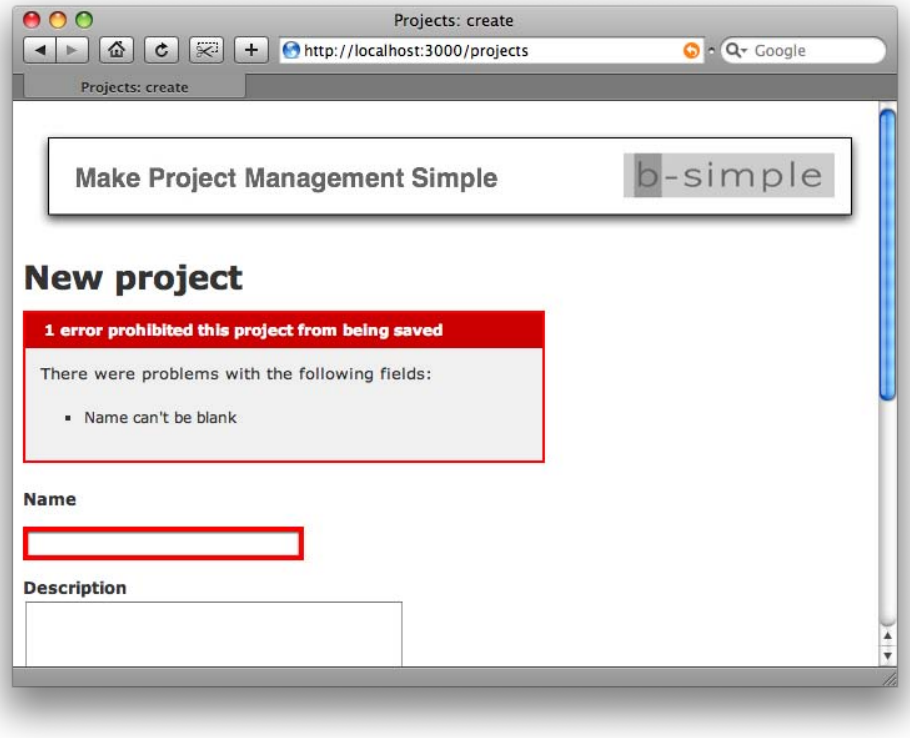

Abbildung 3.11: Projekte ohne Namen sind nicht erlaubt.

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).

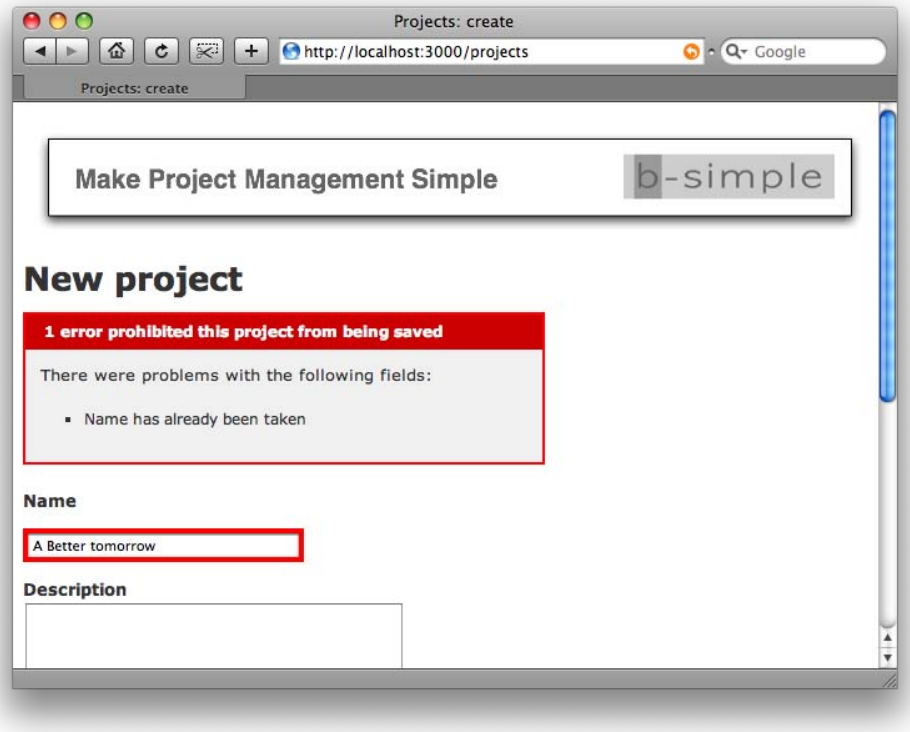

Abbildung 3.12: Projektnamen müssen eindeutig sein.

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
```

[12] Rails 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).

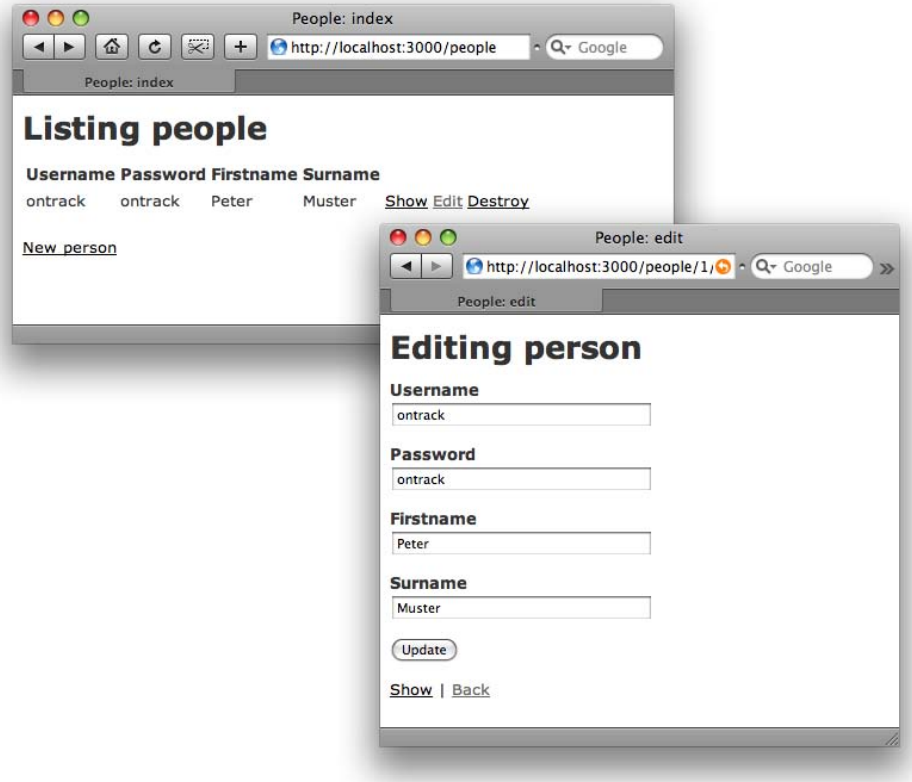

Abbildung 3.13: Die OnTrack-Benutzerverwaltung

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*).

[13] Für 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*s 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 Helpern 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-

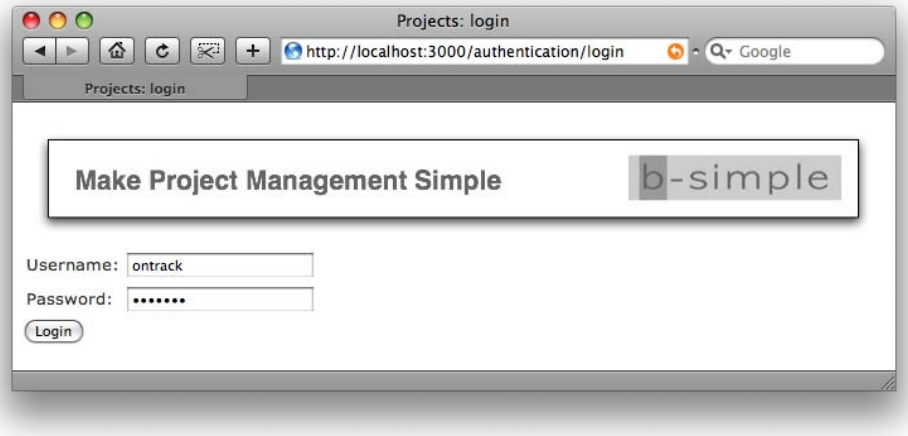

Abbildung 3.14: Die Login-Seite

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)

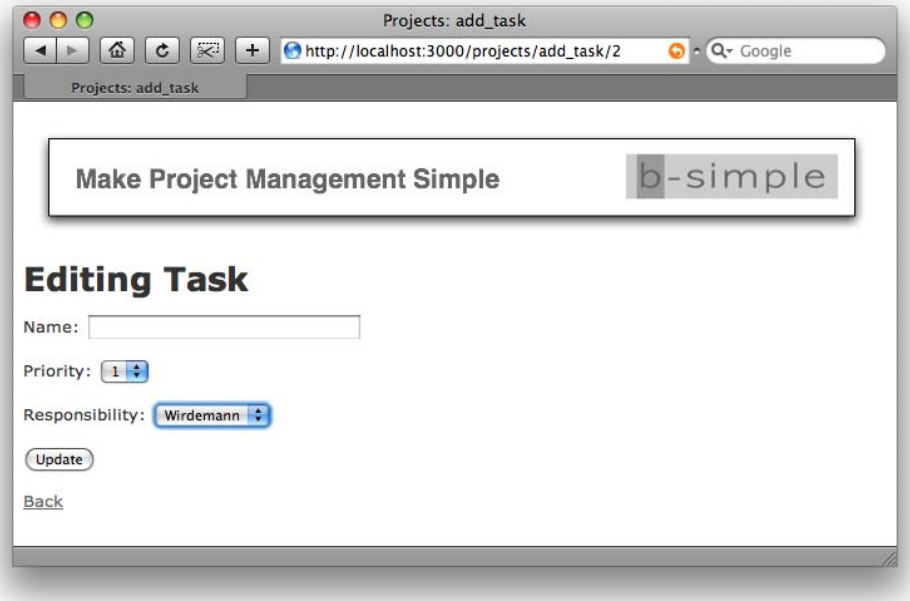

Abbildung 3.15: Zuordnung von Tasks

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.