1 Einstieg

Herzlich willkommen liebe Leserinnen und Leser,

wir freuen uns, dass Sie sich für dieses Rails-Buch entschieden haben. Es beinhaltet eine vollständige Einführung in die Entwicklung von Webapplikationen mit Ruby on Rails (kurz Rails) und gibt Ihnen konkrete Hinweise, wie Sie Herausforderungen in der Webentwicklung lösen können. Rails hat mittlerweile schon einige Jahre auf dem Buckel, seit es am 13. Dezember 2005 in der Version 1.0 als Public-Release das Licht der Software-Welt erblickte. Innerhalb dieser Zeit hat sich das Framework enorm weiterentwickelt, wie auch die Webentwicklung an sich. Durch das geschickte Marketing von Rails ist das Framework den meisten Webentwicklern bereits ein Begriff, selbst wenn sie nicht direkt mit Rails arbeiten. Rails inspirierte ebenso Entwickler abseits der Programmiersprache Ruby. Hierbei entstanden Webframeworks, die Rails sehr ähnlich sind, beispielsweise CakePHP oder Grails aus der Java-Welt. Durch die große Community von Rails entwickelt sich das Framework stetig weiter. Rails ist bequem einsetzbar, um kleine Webseiten oder Portale zu entwickeln, aber auch für die Entwicklung geschäftskritischer Anwendungen verwendbar. Mit diesem Buch möchten wir Ihnen zeigen, wie Sie mit Rails Webanwendungen entwickeln können. Wenn Sie bereits Erfahrungen mit Rails sammeln konnten, zeigen wir Ihnen außerdem, was sich in der Version 3.0 von Rails verändert hat.

1.1 Zielgruppe des Buches

Dieses Buch ist für Einsteiger geeignet, die noch keine Erfahrungen mit Rails haben. Aber auch Entwickler, die bereits Rails-Applikationen entwickelt haben, werden auf ihre Kosten kommen. Wollen Sie gerade mit einem IT-Studium/-Ausbildung beginnen oder stecken gar mitten drin? Möchten Sie Rails 3 in einem Projekt einsetzen? Oder haben Sie bereits Applikationen mit Rails 2 entwickelt? Dann ist dieses Buch genau richtig, um Sie an das Thema „Rails“ heranzuführen oder bereits vorhandenes Wissen zu vertiefen. In diesem Buch setzen wir allerdings voraus, dass Sie ein Grundverständnis für die Technologie des Internets mitbringen und wissen, wie man unter Anleitung ein Programm installiert. Vorteilhaft wäre ebenfalls, wenn Sie bereits Erfahrungen mit einer Programmiersprache, speziell mit einer Skriptsprache wie zum Beispiel PHP, Perl oder Python haben. Grundlagenwissen in der Programmiersprache Ruby wäre optimal, wird für das Verständnis des Buches aber nicht zwingend vorausgesetzt, da alles Nötige Stück für Stück erklärt wird.

1.2 Informationen/Feedback

Die Webseite zu diesem Buch finden Sie unter *http://rails.spider-network.net*. Auf dieser Webseite finden Sie Korrekturen und weiterführende Informationen zu diesem Buch sowie zum Thema Rails selbst. Bitte senden Sie uns Ihre Anmerkungen, Hinweise und Korrekturvorschläge an rails-book@spider-network.net. Informationen zum Verlag und dessen Verlagsprogramm finden Sie unter *http://www.entwickler-press.de*.

1.3 Aufbau des Buches

Hier möchten wir Ihnen einen kurzen Ausblick über den Inhalt und den Aufbau des Buches geben. Die Besonderheit des Buches besteht darin, dass die meisten Quelltextbeispiele einer laufenden Applikation (*http://www.kraeftemessen.com*) entnommen wurden und ihr gesamter Quelltext auf Github (*http://github.com/spider-network/kraeftemessen*) einsehbar ist. Bei den jeweiligen Kapiteln wird diese Applikation als Leitfaden für die Erläuterung der verschiedenen Themengebiete verwendet. Bei der Applikation handelt es sich um eine Veranstaltungsdatenbank für sportliche Ausdauerwettkämpfe, mit der Besucher nach Veranstaltungen aus Ihrer Umgebung suchen können.

Kapitel 1 – Einstieg

In diesem Kapitel werden wir zum einen auf den allgemeinen Aufbau von Rails und zum anderen auf die geschichtliche Entwicklung des Frameworks eingehen. Des Weiteren werden wir in diesem Kapitel eine minimale Rails-Applikation erstellen, die bereits mit der Datenbank kommuniziert. Anhand dieser kleinen Beispielapplikation wollen wir die einzelnen Komponenten von Rails kurz anreißen und dann in den nachfolgenden Kapiteln konkreter erläutern. Das erste Kapitel soll uns somit ein vollständiges Bild von Rails vermitteln, ohne auf spezielle Details der einzelnen Komponenten einzugehen.

Kapitel 2 – ORM-Bibliotheken (Model-Komponente)

Die Model-Komponente stellt das „M“ im MVC-Framework dar. Innerhalb dieses Kapitels werden wir auf das Thema ORM-Bibliothek eingehen sowie seine Einsatz- und Verwendungsmöglichkeiten erläutern. In Ruby gibt es eine Vielzahl von ORM-Bibliotheken wie beispielsweise Sequel, DataMapper und ActiveRecord. Wir werden in diesem Kapitel detaillierter auf die ORM-Bibliothek DataMapper eingehen. Die übrigen Beispiele im Buch verwenden die ORM-Bibliothek ActiveRecord.

Kapitel 3 – Template (View-Komponente)

Die View-Komponente stellt das „V“ im MVC-Framework dar. In diesem Kapitel gehen wir primär auf die Template-Sprache Haml für die Generierung von HTML und auf die Metasprache Sass für das Generieren von CSS ein.

Kapitel 4 – Programmlogik (Controller-Komponente)

Neben dem „M“ und dem „V“ im MVC-Framework fehlt uns nun nur noch das „C“, die Controller-Komponente. In diesem Kapitel gehen wir darauf ein, wie eine HTTP-Anfrage von der Rails-Applikation konkret verarbeitet wird. Auch weiterführende Themen wie die Ruby-Webserver-Schnittstelle Rack und der Softwarearchitekturstil „Representational State Transfer“, der unter der Abkürzung REST bekannt ist, finden in diesem Kapitel ihre Berücksichtigung.

Kapitel 5 – E-Mail-Nachrichten verarbeiten (ActionMailer)

Mit dem ActionMailer können wir E-Mail-Nachrichten unter Rails sowohl versenden als auch empfangen. In Rails 3 wurde diese Komponente vollständig überarbeitet und die Handhabung deutlich vereinfacht.

Kapitel 6 – Testen

Im Kapitel „Testen“ gehen wir auf die Grundaspekte des Schreibens von automatisierten Tests ein. Darüber hinaus stehen aber auch Themen wie die Nutzung und die Installation eines Continuous Integration Servers (kurz CI-Server) innerhalb dieses Kapitels im Vordergrund.

Kapitel 7 – Internationalisierung und Lokalisierung

Beim Thema Internationalisierung und Lokalisierung handelt es sich um ein sehr umfangreiches Themengebiet. Wir werden in diesem Kapitel zeigen, welche Möglichkeiten Rails standardmäßig bietet, um Applikationen, die mit dem Framework entwickelt wurden, zu internationalisieren bzw. zu lokalisieren.

Kapitel 8 – Werkzeugkasten

In diesem Kapitel zeigen wir, wie Sie konkrete Herausforderungen aus der Softwarewelt am besten lösen können. Themen wie das Erstellen einer einfachen Blätterfunktion oder eines PDF-Dokuments, aber auch Themen wie das Durchsuchen von großen Datenmengen mithilfe einer Full-Text Search Engine werden in diesem Kapitel näher erläutert. Alle Leser, die bereits Erfahrung mit Ruby und dem Framework Rails haben, werden in diesem Kapitel sicherlich noch einige gute Anregungen bzw. Tipps finden.

1.4 Voraussetzungen

Dieses Buch erläutert die einzelnen Themengebiete anhand von zahlreichen praxisorientierten Beispielen. Für ein besseres Verständnis empfehlen wir Ihnen, einzelne Passagen abzutippen und dabei auch gern einen Schritt weiter zu gehen als im Buch, frei nach dem Motto „learning by doing“. Selbstverständlich werden ausgewählte Quelltextbeispiele auf der Webseite zum Buch zu Ihrer Verfügung bereitgestellt. Sie benötigen einen Computer, auf dem die Programmiersprache Ruby sowie der zugehörige Paketmanager RubyGems installiert sind. In diesem Buch sind die meisten Beispiele mit einer MySQL-5-Datenbank im Hintergrund umgesetzt. Hierbei ist es irrelevant, ob Sie als Betriebssystem

Windows, Linux oder Mac OS X verwenden. Die Programmiersprache Ruby ist betriebssystemunabhängig. Wir persönlich bevorzugen Mac OS X, was an einigen Stellen des Buches durchaus sichtbar sein kann.

Bei konkreten Fragen können Sie uns jederzeit eine E-Mail schreiben. Gegebenenfalls werden wir Fragen zusammenfassend auf der Webseite zum Buch beantworten.

1.5 Ruby on Rails

Bis jetzt ist der Begriff „Rails“ schon ein dutzend Mal gefallen. Aber was ist Rails eigentlich genau? Bei Rails handelt es sich um ein Open-Source-Webframework, welches vom dänischen Programmierer David Heinemeier Hansson entwickelt wurde. Er extrahierte Rails aus seiner Arbeit am Projektmanagementtool Basecamp und präsentierte es Mitte 2004 das erste Mal der Öffentlichkeit. Das Webframework Rails basiert auf der „Model View Controller“-Architektur (kurz MVC). Die MVC-Architektur unterteilt die eigene Softwarestruktur in eine Model- (Model), Präsentations- (View) und Steuerungsschicht (Controller). Wie der Name bereits erkennen lässt, ist Rails in der Programmiersprache Ruby geschrieben. Im Laufe der Entwicklung von Rails ist ein modulares Webframework entstanden. So legt uns Rails nicht auf eine bestimmte Template-Sprache (z. B. ERB, Haml), JavaScript-Bibliothek (z. B. jQuery, Prototype) oder einen bestimmten OR-Mapper (z. B. ActiveRecord, Sequel, DataMapper) fest.

Rails unterliegt dem Prinzip „Convention over Configuration“, das heißt, dass Rails gewisse Konventionen vorgibt. Beispielsweise entspricht der Tabellenname in der Datenbank dem Plural des Modelnamens, wenn wir den Tabellennamen im Model nicht explizit über die Methode *set_table_name* setzen. In verschiedenen anderen Frameworks zur Erstellung von Software ist es üblich, zahlreiche Dinge in Konfigurationsdateien vorkonfigurieren zu müssen, bevor man erste Ergebnisse sehen konnte. Dies ist bei Rails nicht der Fall, vorausgesetzt, wir folgen den Konventionen, die Rails vorgibt. Hierdurch sind eine beschleunigte Softwareentwicklung und rasche Umsetzung von Ideen möglich.

Des Weiteren wurde die Webentwicklung von Rails durch den Leitsatz „Don’t Repeat Yourself“ (DRY) geprägt. Das bedeutet, dass der Quelltext einer bestimmten Funktionalität nicht in mehrere Methoden dupliziert wird und somit Wiederholungen vermieden werden. Rails prägte die Welt der Webentwicklung ebenfalls im Bereich des Testens von Applikationen, da Rails von Haus aus ein gut integriertes und somit bequem nutzbares Testframework beinhaltet. Das gibt dem Entwickler die Möglichkeit, den Test noch vor der eigentlichen Implementierung zu schreiben. Dieser Entwicklungstil wird auch als „Test-driven Development“ (kurz TDD) bezeichnet.

1.5.1 Historische Entwicklung

Rails hat sich von 2005 bis zum heutigen Tag enorm weiterentwickelt. Lassen Sie uns nun kurz die genaue Entwicklung von Rails nachvollziehen.

Version 1.0

Im Dezember 2005 wurde die Version 1.0 fertiggestellt. Ziel dieser Version war es, Rails nach seiner Extraktion aus Basecamp auf ein solides Grundfundament zu setzen, sodass Webanwendungen für den produktiven Einsatz mithilfe von Rails entwickelt werden konnten.

Version 1.1

Bereits vier Monate später, im März 2006, wurde die Version 1.1 veröffentlicht. Neben den mehr als 500 Bugfixes wurden auch neue Funktionen wie RJS-Templates hinzugefügt. RJS-Templates erzeugen JavaScript-Quelltext, der ausgeführt wird, sobald dieser an den Browser zurückgegeben wird. Weiterhin wurden beim OR-Mapper ActiveRecord einige Veränderungen vorgenommen. Es war nun möglich, M:N-Beziehungen über eine konkrete Model-Klasse zu modellieren (*has_many :friends, :through => :friendships*). Auch die polymorphe Beziehung wurde neu eingeführt. Hiermit können wir ausdrücken, dass die Model-Klassen User und Event dasselbe Address-Model verwenden, sodass es in der Datenbank nur eine Tabelle für die Adressen gibt. Für eine bessere Performance bei der Nutzung von ActiveRecord wurde das JOIN-ing zwischen Model-Klassen optimiert. Über die Funktionalität von *respond_to* war es jetzt möglich, im Controller besser auf Anfragen mit verschiedenen Content-Typen zu reagieren und somit unterschiedliche Formate als Antwort zurückzuliefern.

Version 1.2

Ein dreiviertel Jahr später, im Januar 2007, erschien die Version 1.2. Die Hauptneuerung in dieser Version war die Integration des Softwarearchitekturstils REST (Representational StateTransfer). Auf diese Technologie werden wir im Kapitel „Programmlogik (Controller-Komponente)“ genauer eingehen. Darüber hinaus wurden zahlreiche Methoden des Frameworks als deprecated (dt. veraltet) gekennzeichnet, die in Rails 2.0 endgültig gelöscht wurden. Durch die starke Verbreitung von Rails über den englischsprachigen Raum hinaus, wurde auch die Behandlung von UTF-8-Zeichenketten verbessert.

Version 2.0

Knapp ein Jahr Entwicklungszeit ging ins Land, bis die Version 2.0 im Dezember 2007 fertig gestellt wurde. Das Major Release 2.0 von Rails brachte weit über einhundert Neuerungen und Verbesserungen mit sich. Neben der Migration von SOAP zu REST wurden auch weitere Funktionen wie die Authentifizierung über HTTP-Basic-Authentication integriert. Die Syntax für die Schreibweise von Migrationen wurde mehr in Richtung DRY verändert (Sexy migrations). Das Referenzieren zwischen verschiedenen Fixtures wurde ebenfalls deutlich vereinfacht (Foxy fixtures).

Version 2.1

Rails 2.1 erschien im Juni 2008. Neuerung in dieser Version war unter anderem die verbesserte Berücksichtigung verschiedener Zeitzonen. Über die Funktionalität „dirty object“ von ActiveRecord hat man vor dem Abspeichern eines Model-Objektes die Möglichkeit, Veränderungen des Objekts anzeigen zu lassen. Mithilfe der Deklaration (*config.gem("haml", :version => "2.2.15")*) in der Datei *config/environment.rb* können wir die Abhängigkeit von der Rails-Applikation zu den verwendeten GEM-Bibliotheken definieren. Häufig verwendete ActiveRecord-Finder-Definitionen können wir nun in *named_scope*-Deklarationen der Model-Klasse kapseln. Eine weitere große Verbesserung war die Umstellung der Versionsnummern der Datenbankmigrationen auf Timestamps, da es zuvor beim Arbeiten im größeren Team immer zu Versionskonflikten gekommen war. Des Weiteren wurden noch einige Optimierungen bei der Caching-Implementierung vorgenommen.

Version 2.2

Im November 2008 wurde die Rails-2.2-Version freigegeben. Seit dieser Version ist Rails mit Ruby 1.9.x und JRuby kompatibel. Die Hauptneuerung in dieser Version war die Integration der I18n-Gem-Bibliothek. Seitdem stellt Rails ein Standard-API für die Internationalisierung bereit. Für das Cachen auf HTTP-Ebene unterstützt Rails 2.2 etag und last-modified im HTTP-Header. Die Thread-Sicherheit von Rails wurde ebenfalls verbessert und ActiveRecord bekam einen „database connection pool“.

Version 2.3

Rails 2.3 erschien im März 2009. Der Umgang mit verschachtelten Model-Klassen wurde dank „nested forms“ stark vereinfacht. Die wohl größte und bedeutendste Änderung dieser Version war die Integration der Webserverschnittstelle Rack. Weiterhin wurde das Modul Rails Metal eingeführt. Mithilfe von Rails Metal ist es möglich, Requests zu beantworten, bevor sie durch den gesamten Stack von Rails gelaufen sind. Dies verringert die Antwortzeit eines HTTP-Requestes deutlich.

Version 3.0

Begonnen hat die Planung für die Version 3.0 bereits im Dezember 2008, als David Heinemeier Hansson am 23. Dezember 2008 ankündigte, dass die besten Funktionen von Rails und dem anderen großen Ruby-Webframework Merb (siehe nächster Abschnitt) zu Rails 3.0 zusammengefügt werden sollen. Hauptkritikpunkt an Rails war bis zu dieser Zeit, dass es nicht gut skaliert und nicht ausreichend modular aufgebaut war. Fest steht, dass Rails 3.0 erst durch den Einfluss von Merb an Modularität und Performance gewonnen hat. Als größte Neuerungen in dieser Version sind die Möglichkeiten der freien Wahl des OR-Mappers (z. B. ActiveRecord, Sequel, DataMapper), der Template-Sprache (z. B. ERB, Haml) und der JavaScript-Bibliothek (z. B. jQuery, Prototype) zu nennen. Weitere neue Funktionalitäten in Rails 3.0 sind unter anderem der Bundler für Gem-Bibliotheken, der neue Router und die Arel-Syntax bei ActiveRecord.

HINWEIS: Merb ist wie Rails ein MVC-Web Framework, welches ebenfalls in Ruby geschrieben ist. Die Geschichte von Merb begann am 22. September 2006, als Ezra Zygmuntowicz auf pastie.org den ersten Quelltext veröffentlichte (siehe Anhang A.1). Er hatte das Problem, dass seine Rails-Applikation blockiert war, während größere Dateien hochgeladen wurden. Dies wollte er mithilfe eines kleinen Frameworks lösen. Der Name Merb entstand aus der Kombination Mongrel, den er als HTTP-Server nutzte, und ERB, welches er als Template-System einsetzte. Er war natürlich nicht der Einzige, der zuvor beschriebenes Problem hatte, sodass auch viele andere Rails-Nutzer bemerkten, dass Rails für einige Anwendungsfälle zu schwergewichtig war. Somit entstand im Laufe der Jahre ein Webframework, bei dem das Augenmerk stark auf Performance und Skalierbarkeit liegt. Weitere Informationen finden Sie auf der offiziellen Webseite von Merb (*http://www.merbivore.com*).

1.5.2 Die verschiedenen Infoquellen

Rails

- Offizielle Webseite: *http://www.rubyonrails.org*
- Offizieller Blog: *http://weblog.rubyonrails.org*
- Was ist neu in Rails (Edge): *http://edgerails.info*
- Wiki: *http://wiki.rubyonrails.org*
- Rails-Guide: *http://guides.rubyonrails.org*
- IRC: irc.freenode.net#rubyonrails
- API-Dokumentation: *http://api.rubyonrails.org*
- Metablog: *http://planetrubyonrails.com*
- Mailingliste (EN): *http://groups.google.com/group/rubyonrails-talk*
- Mailingliste (DE): *http://www.rubyonrails-ug.de*

Ruby

- Offizielle Ruby-Webseite: *http://www.ruby-lang.org*
- Dokumentation: *http://ruby-doc.org*

1.5.3 Komponenten

Den kompletten Funktionsumfang der einzelnen Komponenten können Sie ebenfalls in der Rails-API-Dokumentation nachlesen.

ActionPack

Die Komponente ActionPack hat in Rails eine zentrale Aufgabe, da sie den Request über den Router annimmt und ihn auf den dazugehörigen Controller und eine Action weiterleitet. Nach der Verarbeitung des Requests durch den Controller gibt ActionPack die Antwort an den Client zurück. Dazu spricht ActionPack die benutzte Template Engine (z. B. Haml, ERB) an und gibt den Inhalt der fertig gerenderten Templates an den Client zurück. Die Komponente ActionPack implementiert somit sowohl die Bestandteile für die Controller-Schicht als auch für die View-Schicht der MVC-Architektur.

ActiveRecord

ActiveRecord stellt in der MVC-Architektur das „M“ (Model) dar. Es ist wie DataMapper oder Sequel ein OR-Mapper und bietet uns eine Abstraktionsschicht zur Datenbank. Das Subframework ActiveRecord ist nach einem Entwurfsmuster (engl. Design Pattern) benannt, welches Martin Fowler in seinem Buch „Patterns of Enterprise Application Architecture“ genau erläutert.

ActiveResource

In Rails 2.0 wurde ActionWebService durch die Komponente ActiveResource ersetzt. Mithilfe von ActiveResource können REST-konforme Webservices (Representational State-Transfer) in Applikationen eingebaut und konsumiert werden.

ActiveModel

Mithilfe der Komponente ActiveModel möchte man vermeiden, dass die Entwicklung einzelner Model-Funktionalitäten wie Observer, Callbacks, Validations oder Serialization auseinanderlaufen. Durch Extraktion von ActiveModel ist es möglich, diese Funktionalität identisch für ActiveRecord und ActiveResource bereitzustellen. Weiterhin können wir diese herausgelösten Funktionen auch in eigene Klassen integrieren.

ActiveSupport

ActiveSupport ist eine Sammlung von nützlichen Klassen und Erweiterungen der Standard-Ruby-Bibliothek, mit denen sich Standard-Programmieraufgaben eleganter lösen lassen. Um die hilfreichen Erweiterungen auch in anderen Bibliotheken nutzen zu können, wurden diese in eine eigene Bibliothek gekapselt. Zahlreiche Subframeworks (z. B. ActiveRecord, ActiveResource) von Rails weisen deshalb eine Abhängigkeit zu dieser Bibliothek auf. ActiveSupport stellt uns beispielsweise zahlreiche Datumshilfsfunktionen bereit. So liefert z. B. die Instanzmethode *end_of_month* eines Date-Objekts den letzten Tag des Monats zurück (*Date.today.end_of_month => Mon, 31 May 2010*).

ActionMailer

ActionMailer ist ein Subframework von Rails, das uns einen abstrahierten Layer für das Senden, Empfangen und Verarbeiten von E-Mails bereitstellt.

Railties

Railties verbindet die vorgenannten Komponenten zu einem Webframework, das wir insgesamt als Rails bezeichnen. Die Komponente Railties umfasst neben den zahlreichen Generatoren für das Anlegen von Controllern, Migrationen, Model-Klassen und dem Generieren von neuen Rails-Applikationen auch das Konfigurationsmanagement der einzelnen Komponenten.

1.6 Lernen am Beispiel

Ganz nach dem Leitsatz „Learning by doing" wollen wir Ihnen die Funktionsweise von Rails anhand einer sehr einfachen Applikation vermitteln, sodass Sie einen Gesamtüberblick über das Webframework Rails erhalten. Die Rails-Applikation Kraeftemessen.com (Veranstaltungsdatenbank) werden wir erst in den Folgekapiteln als Beispiel für die jeweiligen Programmierthematiken heranziehen. Für die erste Rails-Applikation haben wir uns überlegt, eine einfache Einkaufsliste (engl. shopping list) zu entwickeln, bei der wir Positionen hinzufügen, aktualisieren und löschen können. Wir denken, dass dies eine sehr einfache und klare Aufgabenstellung ist, die wir nun gemeinsam umsetzen wollen.

1.6.1 Installation

Da es schwierig ist, eine allgemeingültige Installationsanleitung zu verfassen, möchten wir Ihnen an dieser Stelle lediglich die wesentlichen Anhaltspunkte für die Installation Ihrer lokalen Entwicklungsumgebung geben. Kommen Sie an einem bestimmten Punkt der Installation nicht weiter oder haben dabei ein Problem, dann können Sie uns natürlich jederzeit kontaktieren. Gegebenenfalls werden wir erweiterte Informationen auf der Webseite zum Buch bereitstellen. Oder Sie verwenden eine von zahlreichen im Internet verfügbaren Installationsanleitungen. Befragen Sie hierfür einfach eine Suchmaschine wie bing.de oder google.de (z. B. „installation rails windows").

Microsoft Windows

Die Installation von Rails und dessen Komponenten unter Microsoft Windows unterscheidet sich deutlich von anderen Betriebssystemen. Zuerst müssen wir Ruby und den dazugehörigen Paketmanager RubyGems installieren. Hierfür steht uns ein One-Click Installer zur Verfügung, den wir von der Webseite *http://www.ruby-lang.org/de/downloads* herunterladen können. Das Installationsskript installiert Ruby standardmäßig im Verzeichnis *c:\ruby*. Vergewissern Sie sich, dass sich der Pfad *c:\ruby\bin* im Suchpfad PATH Ihrer Umgebung befindet, ansonsten können Sie die nachfolgenden Befehle nicht auf der Kommandozeile ausführen. Sie können dies über den Befehl *path* auf der Kommandozeile überprüfen.

Lassen Sie uns als Erstes den Paketmanager RubyGems aktualisieren.

```
01| $> gem update --system
```

Anschließend installieren wir das Webframework Rails über den Paketmanager RubyGems

```
01| $> gem install rails --pre
```

Wenn Sie nicht bereits einen MySQL-Server auf Ihrem Computer installiert haben, laden Sie sich bitte das Installationsskript von der MySQL-Webseite *http://dev.mysql.com/downloads/mysql/5.1.html#win32* herunter und führen danach das Installationsskript aus.

Nachdem wir den MySQL-Server installiert haben, benötigen wir die Ruby-Bibliothek für die Kommunikation mit der MySQL-Datenbank, die wir ebenfalls über den Paketmanager RubyGems installieren können.

```
01| $> gem install mysql
```

Sie sollten nun eine funktionsfähige Entwicklungsumgebung für Rails auf Ihrem Computer installiert haben.

Mac OS X

Mac OSX Leopard beinhaltet Rails bereits in der Version 1.2.6 und Ruby in der Version 1.8.6. Rails 3.0 setzt Ruby 1.8.7 oder eine aktuellere Version voraus. Wer bereits Mac OS X in der Version Snow Leopard installiert hat, kann die folgende Installation von Ruby überspringen, da Ruby 1.8.7 bereits installiert ist.

Im Kapitel „Werkzeugkasten" erläutern wir, wie wir mehrere Ruby-Versionen über die Gem-Bibliothek RVM (Ruby Version Manager) verwalten können. Wir können Ruby aber auch über den Paketmanager MacPorts (*http://www.macports.org*) installieren, oder Ruby wie folgt selber kompilieren.

```
01| $> wget ftp://ftp.ruby-lang.org/pub/ruby/1.9/ruby-1.9.1-p376.tar.gz
02| $> tar xzvf ruby-1.9.1-p376.tar.gz
03| $> cd ruby-1.9.1-p376
04| $> ./configure
05| $> make
06| $> sudo make install
```

- **Zeile 01:** Sollten Sie *wget* nicht installiert haben, können Sie sich den Quelltext von der Webseite *http://www.ruby-lang.org/de/downloads/* herunterladen.

Der Paketmanager RubyGems wird wie folgt aktualisiert:

```
01| $> sudo gem update --system
```

Die aktuelle Version von Rails installieren wir über den Paketmanager RubyGems.

```
01| sudo gem install rails --pre
```

Wenn Sie nicht bereits einen MySQL-Server auf Ihren Rechner installiert haben, laden Sie sich bitte das Installationsskript von der MySQL-Webseite *http://dev.mysql.com/downloads/mysql/5.1.html#macosx-dmg* herunter und führen danach das Installationsskript aus.

Nachdem wir den MySQL-Server installiert haben, benötigen wir die Ruby-Bibliothek mysql für die Kommunikation mit der MySQL-Datenbank, die wir ebenfalls über den Paketmanager RubyGems installieren können.

```
01| $> sudo env ARCHFLAGS="-arch x86_64" gem install mysql -- \
02|         --with-mysql-dir=/usr/local/mysql \
03|         --with-mysql-lib=/usr/local/mysql/lib \
04|         --with-mysql-include=/usr/local/mysql/include
```

Die verwendeten Pfade können bei Ihrer MySQL-Installation abweichen. Abhängig von der Art der Installation (32 oder 64 bit), variiert auch der Wert des Parameters „ARCHFLAGS“.

Sie sollten nun eine funktionsfähige Entwicklungsumgebung für Rails auf Ihrem Computer installiert haben.

Linux

Die Installationsschritte hängen stark von der gewählten Distribution (z. B. Ubuntu, Debian, CentOS) ab. Am bequemsten ist die Installation der Komponenten über den Paketmanager der gewählten Distribution.

Finden wir keine fertigen Installationspakete, haben wir die Möglichkeit, Ruby selber zu kompilieren. Die aktuelle Ruby-Version finden wir auf der Webseite *http://www.ruby-lang.org/de/downloads/* zum Herunterladen.

```
01| $> wget ftp://ftp.ruby-lang.org/pub/ruby/1.9/ruby-1.9.1-p376.tar.gz
02| $> tar xzvf ruby-1.9.1-p376.tar.gz
03| $> cd ruby-1.9.1-p376
04| $> ./configure
05| $> make
06| $> sudo make install
```

Ruby in der Version 1.9.1 beinhaltet bereits den Paketmanager RubyGems, sodass wir Rails direkt über den Paketmanager installieren können.

```
01| $> sudo gem install rails --pre
```

Wenn Sie nicht bereits einen MySQL-Server auf Ihren Rechner installiert haben, stehen auf der MySQL-Webseite *http://dev.mysql.com/downloads/mysql/5.1.html* für einige Distributionen fertige RPM-Pakete bereit.

Nach der Installation des MySQL-Servers benötigen wir die Ruby-Bibliothek mysql für die Kommunikation mit der MySQL-Datenbank, die wir ebenfalls über den Paketmanager RubyGems installieren können.

```
01| $> gem install mysql
```

Sie sollten nun eine funktionsfähige Entwicklungsumgebung für Rails auf Ihrem Computer installiert haben.

1.6.2 Eine Rails-Applikation erstellen

Lassen Sie uns nach der erfolgreichen Installation von MySQL, Ruby, RubyGems und Rails damit beginnen, eine erste Rails-Applikation zu generieren. Hierfür stellt uns Rails das Kommandozeilentool *rails* bereit. Über die Option *help* bekommen wir eine kurze Dokumentation der jeweiligen Optionen angezeigt (*rails --help*). Standardmäßig wird eine Rails-Applikation mit der Konfiguration für eine SQLite-Datenbank generiert. Beim Erstellen der Rails-Applikation können wir über die Option *database* angeben, welche Datenbank wir verwenden möchten. Zur Auswahl steht uns neben dem Standardwert *sqlite3* auch *mysql*, *oracle*, *postgresql*, *frontbase* und *ibm_db*. Lassen Sie uns nun wie folgt unsere erste Rails-Applikation erstellen:

```
01| $> rails new shopping_list --database=mysql
02|       create
03|       create  README
04|       create  Rakefile
05|       create  config.ru
06|       create  Gemfile
07|       create  app
08|       create  app/controllers/application_controller.rb
09|       ...
```

Anschließend befindet sich im aktuellen Arbeitsverzeichnis ein neues Verzeichnis *shopping_list* mit der neuen Rails-Applikation.

1.6.3 Verzeichnisstruktur von Rails

Rails erzeugt uns beim Erstellen einer neuen Rails-Applikation eine Vielzahl von Unterverzeichnissen. In der folgenden Tabelle wollen wir Ihnen die Bedeutung der einzelnen Verzeichnisse erläutern.

Verzeichnis	Beschreibung
app	Stellt das Herzstück der Applikation dar. In diesem Verzeichnis befinden sich standardmäßig die Controller, Helper, Model-Klassen und Views der Applikation.
app/controllers	In diesem Verzeichnis liegen die Controller der Applikation, die den Programmfluss der Applikation steuern.
app/helpers	Hier haben wir die Möglichkeit, Helper-Methoden abzulegen, die wir dann in den Views verwenden können.

Verzeichnis	Beschreibung
app/models	In diesem Verzeichnis liegen die Model-Klassen des OR-Mappers, welche die Schnittstelle zur Datenbank repräsentieren und die Businesslogik der Applikation beinhalten.
app/views	Hier liegen standardmäßig die Template-Dateien.
config	Hier befinden sich die Konfigurationsdateien von Rails. Diese reichen von der Deklaration der Datenbankverbindung bis zur Konfiguration des Debug-Levels in den Logdateien.
db	In diesem Verzeichnis liegen die Migrationsdateien für die Manipulation des Datenbankschemas.
doc	Verzeichnis für verschiedene Dokumentationen.
lib	Hier ist Platz für das Schreiben von eigenen Rake Tasks und Bibliotheken.
log	In dieses Verzeichnis werden alle Logdateien von Rails geschrieben.
public	In diesem Verzeichnis liegen statische Dateien wie CSS-Stylesheets, JavaScript-Dateien und HTML-Fehlerseiten (404-, 500-Seite), die direkt vom Webserver ausgeliefert werden können.
script	In diesem Verzeichnis liegt die Skript-Datei *rails*, welche uns unter anderem folgende Unterkommandos bereitstellt: *rails server*, *rails dbconsole* und *rails console*.
test	Hier sind die Testdateien der unterschiedlichen Testsuiten abgelegt.
tmp	Dieses Verzeichnis wird für temporären Inhalt verwendet.
vendor	In diesem Verzeichnis liegen externe Bibliotheken, die für die eigene Applikation benötigt werden, wie beispielsweise Plug-ins und Gem-Bibliotheken.

1.6.4 Konfiguration

Wie uns der Leitslogan „Convention over Configuration“ von Rails bereits verdeutlicht, müssen wir nicht zahlreiche Konfigurationsparameter setzen, um eine Rails-Applikation zu starten. Trotzdem können wir bei Rails das Standardverhalten derart konfigurieren, dass das Framework besser auf unsere Bedürfnisse angepasst ist. Wir können zum Beispiel die Ladereihenfolge der installierten Plug-ins, den Verzeichnispfad zu den Templates, aber auch den verwendeten OR-Mapper der Rails-Applikation bestimmen. Rails stellt uns unterhalb des Verzeichnisses config eine Reihe von Dateien für die verschiedenen Parameter der Konfiguration bereit.

config/database.yml

Sobald eine Rails-Applikation mit einer Datenbank kommunizieren soll, müssen wir die Zugangsdaten für die Verbindung zur Datenbank in der Datei *database.yml* konfigurieren. Die Konfigurationsdatei könnte für eine MySQL-Datenbank wie in der folgenden Quelltextabbildung dargestellt aussehen. Beim Erstellen einer Rails-Applikation wird ein Konfigurations-Template für die verwendete Datenbank angelegt. Wie zuvor erläutert, können Sie über die Option *database* bei der Generierung einer Rails-Applikation, die zu verwendende Datenbank angeben (*rails new shopping_list –-database=mysql*).

```
01| default: &defaults
02|   adapter: mysql
03|   encoding: utf8
04|   reconnect: false
05|   pool: 5
06|   username: spider-network-mysql
07|   password: mein_passwort
08|   socket: /tmp/mysql.sock
09|
10| development:
11|   <<: *defaults
12|   database: shopping_list_development
13|
14| test:
15|   <<: *defaults
16|   database: shopping_list_test
17|
18| production:
19|   <<: *defaults
20|   database: shopping_list_production
21|   password: ein_anderes_passwort
```

- **Zeile 01-08:** Hier fassen wir die Gemeinsamkeiten der Datenbankkonfiguration für die einzelnen Umgebungen (development, test, production) in einem Schlüssel mit dem Namen default zusammen. Über den Aufruf *<<: *defaults* mixen wir diese Konfiguration in die jeweilige Konfiguration der Umgebung. Dies können Sie in Zeile 11, 15 und 19 sehen.
- **Zeile 10-21:** Hier konfigurieren wir die Datenbankverbindung für die Development-, Test- und Produktionsumgebung, wobei wir für die Produktionsumgebung ein anderes Passwort verwenden.

config/application.rb

In der Konfigurationsdatei *application.rb* werden Konfigurationsparameter gesetzt, die in allen Umgebungen (Development, Test, Production) identisch sein sollen. Dies betrifft zum Beispiel die Deklaration der verwendeten Template-Sprache oder des OR-Mappers.

```
01| module ShoppingList
02|   class Application < Rails::Application
03|     config.plugins = [ :exception_notification, :all ]
04|
05|     config.time_zone = 'Berlin'
06|
07|     config.generators do |g|
08|       g.orm :active_record
09|       g.template_engine :erb
10|       g.test_framework  :test_unit, :fixture => true
11|     end
12|   end
13| end
```

Listing 1.1: config/application.rb

- **Zeile 03:** Über *config.plugins* können wir konfigurieren, in welcher Reihenfolge die Plug-ins geladen werden sollen. In unserem Beispiel laden wir als erstes das Plug-in *exception_notification* und anschließend alle anderen Plug-ins. Dies drücken wir über das Symbol *:all* aus.
- **Zeile 05:** Über *config.time_zone* können wir die Zeitzone für die Applikation konfigurieren.
- **Zeile 07-**11: Über *config.generators* konfigurieren wir, welche Template-Sprache, welcher OR-Mapper und welches Testframework bei der Verwendung der Rails-Generatoren (z. B. *rails generate model user*) angesprochen werden sollen.

config/environments/*.rb (development, test, production)

Weiterhin können wir verschiedene Einstellungen für die einzelnen Umgebungen konfigurieren. Hierfür gibt es unterhalb des Verzeichnisses *config/environments* für die jeweilige Umgebung eine Konfigurationsdatei. Sinnvoll ist es zum Beispiel, in der Development-Umgebung das Cachen von Klassen zu deaktivieren, wodurch wir bei Änderungen am Quelltext nicht jedes Mal den Webserver neustarten müssen.

```
01| config.cache_classes = false
02| ...
```

Listing 1.2: config/environments/development.rb

Wie folgt können wir das Senden von E-Mail-Nachrichten in der Testumgebung unterbinden:

```
01| config.action_mailer.delivery_method = :test
02| ...
```

Listing 1.3: config/environments/test.rb

Diese genannten Konfigurationen sind bei Rails bereits standardmäßig vorkonfiguriert.

Des Weiteren können wir auch verschiedene Backends für das Cache-Systeme konfigurieren, wie zum Beispiel Memcache oder FileStore. Es folgt ein einfaches Beispiel für die Konfiguration des Cache-Systems:

```
01| config.cache_store = :file_store, '/cache_folder'
```

Mit dieser Konfiguration würden wir das Backend FileStore für das Cache-System aktivieren. Zusätzlich deklarieren wir, dass im Cache gespeicherte Werte in individuelle Dateien unterhalb des Verzeichnisses *cache_folder* abgelegt werden.

```
01| config.cache_store = [
02|   :mem_cache_store,
03|   "127.0.0.1:13011",
04|   { :namespace => 'kraeftemessen_production' }
05| ]
```

Mit dieser Konfiguration würden wir das Backend Memcache für das Cache-System aktivieren. Standardmäßig geht Rails davon aus, dass der Memcache-Daemon auf *localhost* unter dem Port 11211 läuft.

config/initializers/*.rb

Rails lädt beim Starten der Applikation alle Dateien, die sich im Verzeichnis *config/initializers* befinden. Die Deklarationen, die wir innerhalb dieser Dateien vornehmen, sind umgebungsunabhängig und somit in jeder Umgebung vorhanden. Die Trennung in unterschiedliche Dateien dient in erster Linie der Übersichtlichkeit, da wir so nicht verschiedene Modulkonfigurationen vermischen.

In der Initializer-Datei *mime_types.rb* können wir der Applikation neue Mime Types hinzufügen.

```
01| Mime::Type.register('application/pdf', :pdf)
```

Listing 1.4: config/initializers/mime_types.rb

Wir können auch benötigte Konfigurationen für einzelne Erweiterungen, wie zum Beispiel für das Plug-in Geokit, über diese Funktionalität deklarieren und beim Starten der Applikation laden.

```
01| if defined? Geokit
02|   GeoKit::default_units   = :kms
03|   GeoKit::default_formula = :flat
04|   Geokit::Geocoders::timeout = 4
05|   Geokit::Geocoders::google = GOOGLE_MAPS_API_KEY
06|   Geokit::Geocoders::provider_order = [:google]
07| end
```

Listing 1.5: config/initializers/geokit_config.rb

config/locals/*.yml

Im Verzeichnis *config/locals* befinden sich die Konfigurationsdateien für die Internationalisierung und Lokalisierung der Applikation. Auf diese werden wir im Kapitel „Internationalisierung und Lokalisierung“ genauer eingehen.

config/routes.rb

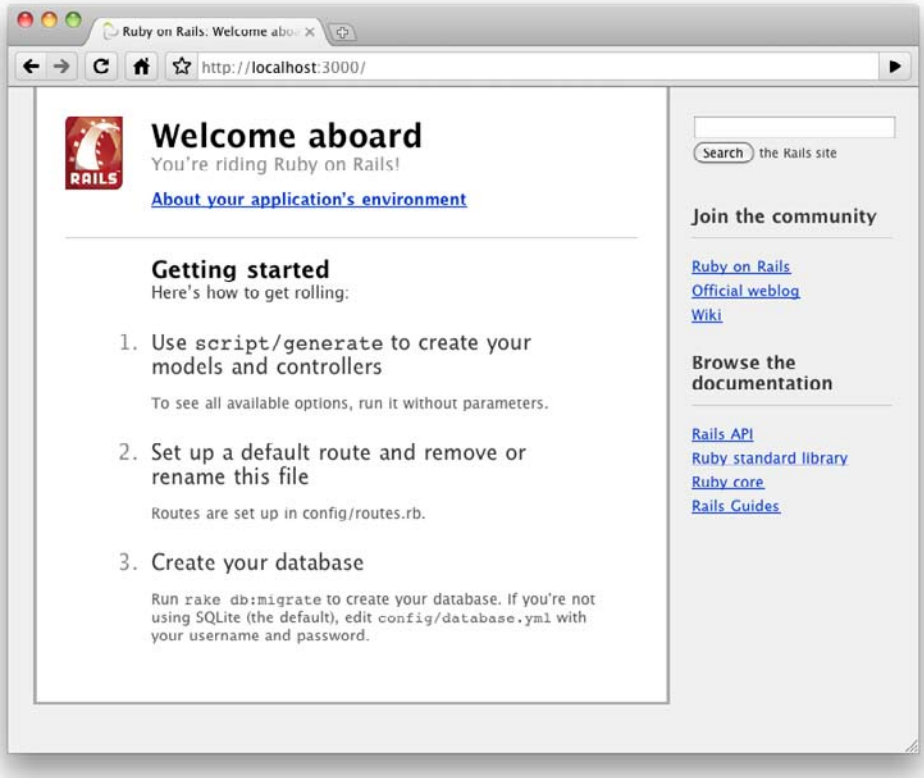
In der Datei *config/routes.rb* können wir das Routing der Applikation steuern, somit festlegen, welcher Applikations-URL von welchem Controller verarbeitet werden soll. Im Kapitel „Programmlogik (Controller-Komponente)“ gehen wir detaillierter auf die Eigenschaften des Router-Moduls ein.

1.6.5 Server starten

Nachdem wir die Rails-Applikation „ShoppingList“ erstellt haben, können wir diese starten und uns das Resultat im Webbrowser ansehen. Den Webserver starten wir mit dem Kommando *rails server* aus dem Verzeichnis unserer Applikation heraus. Ist der Webserver Mongrel installiert, wird dieser standardmäßig als Webserver verwendet, ansonsten wird der Webserver WEBrick genutzt. Mongrel ist im Gegensatz zu WEBrick für das Produktionssystem geeignet.

```
01| $> rails server
02| => Booting Mongrel
03| => Rails 3.0 application starting on http://0.0.0.0:3000
04| => Call with -d to detach
05| => Ctrl-C to shutdown server
```

Der Webserver wird standardmäßig auf dem Port 3000 gestartet. Über den Parameter *-p* können wir einen alternativen Port angeben (*rails server -p5000*). Wenn wir nun im Webbrowser die Adresse *http://localhost:3000* aufrufen, bekommen wir die folgende Seite angezeigt.



Der Webserver liefert die statische HTML-Datei *index.html* aus dem Verzeichnis *public* aus. Löschen wir diese HTML-Datei und rufen im Webbrowser die Adresse erneut auf, bekommen wir folgende Fehlermeldung angezeigt.

Der Grund für diesen Fehler ist, dass die Applikation noch keine Route zu einem entsprechenden Controller besitzt. Somit kann unsere Applikation die HTTP-Anfrage nicht verarbeiten und deshalb auch keine passende Antwort an den Webbrowser zurückliefern.

1.6.6 Der erste Controller

Die grundlegende Verzeichnisstruktur der Applikation „ShoppingList“ haben wir bereits angelegt. Lassen Sie uns im nächsten Schritt einen einfachen Controller erstellen. Dieser soll lediglich einen HTTP-Request entgegennehmen können und eine HTML-Datei als Antwort zurückliefern. Wir werden dies am Beispiel einer einfachen Impressumseite demonstrieren.

Rails bietet uns eine Reihe von Quelltextgeneratoren für das Erstellen von Controllern oder Model-Klassen. Eine vollständige Liste der Quelltextgeneratoren bekommt man angezeigt, wenn man im Verzeichnis der Applikation das Kommando *rails generate* ausführt.

```
01| $> rails generate
02| Please choose a generator below.
03|
04| Rails:
05|   controller
06|   generator
07|   helper
08|   integration_test
09|   mailer
10| migration
11| model
12| ...
```

Eine kurze und übersichtliche Hilfe zu den jeweiligen Generatoren erhalten wir mit dem Parameter *--help* (z. B. *rails generate controller --help*). Neben der Funktionalität *rails generate* gibt es auch das Gegenstück *rails destroy*, mit dessen Hilfe wir den generierten Quelltext vollständig wieder löschen können.

Den Controller für das Impressum legen wir mit folgendem Quelltextgenerator an: *rails generate controller NAME [action action][options]*.

```
01| $> rails generate controller impressum index
02|       create  app/controllers/impressum_controller.rb
03|       invoke  erb
04|       create    app/views/impressum
05|       create    app/views/impressum/index.html.erb
06|       invoke  test_unit
07|       create    test/functional/impressum_controller_test.rb
08|       invoke  helper
09|       create    app/helpers/impressum_helper.rb
10|       invoke    test_unit
11|       create      test/unit/helpers/impressum_helper_test.rb
```

- **Zeile 01:** Hier generieren wir den Controller *ImpressumController* mit der Action *index*.
- **Zeile 02-11:** Zeigt an, welche Verzeichnisse und Dateien vom Quelltextgenerator erstellt wurden.

In unserem Fall müssen wir an der Controller-Datei *app/controllers/impressum_controller.rb* keine Änderungen vornehmen, da wir nur ein statisches HTML-Template ausliefern wollen.

```
01| class ImpressumController < ApplicationController
02|   def index
03|   end
04| end
```

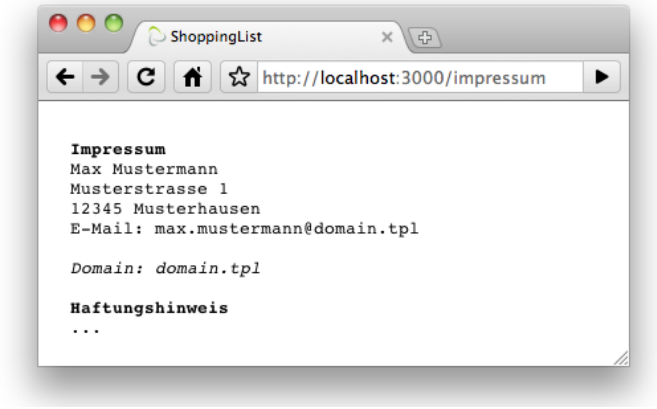
Listing 1.6: app/controllers/impressum_controller.rb

Wenn die Template-Datei denselben Namen wie die Action im Controller aufweist, findet Rails das dazugehörige Template per Konvention selbstständig. Der Quelltextgenerator hat uns bereits eine leere Template-Datei *app/views/impressum/index.html.erb* erstellt, die wir jetzt nur noch auf unsere Bedürfnisse anpassen müssen. Dies könnte wie folgt aussehen:

```
01| <pre>
02|   <b>Impressum</b>
03|   Max Mustermann
04|   Musterstrasse 1
05|   12345 Musterhausen
06|   E-Mail: max.mustermann@domain.tpl
07|
08|   <i>Domain: domain.tpl</i>
09|   ...
10| </pre>
```

Im Kapitel „Programmlogik (Controller-Komponente)“ gehen wir detaillierter auf die Eigenschaften des Routers ein.

Nun starten wir den Webserver mit dem Kommando *rails server* und rufen im Webbrowser die Adresse *http://localhost:3000/impressum/index* auf.



Wenn Sie sich im Webbrowser den HTML-Quelltext der Webseite anzeigen lassen, wird Ihnen auffallen, dass das HTML-Grundgerüst inklusive DOCTYPE-Definition fehlt und somit kein valides HTML-Dokument vorliegt. Wir könnten das HTML-Grundgerüst mit in die Datei *app/views/impressum/index.html.erb* schreiben. Das entspräche aber nicht dem Leitprinzip DRY, da wir dies bei jedem weiteren Template der Applikation wiederholen müssten. Um solche Duplikation zu vermeiden bietet Rails bietet uns die Funktionalität der Layouts, die im Verzeichnis *app/views/layouts/* abgelegt werden. Das Layout mit dem Namen application (z. B. *application.html.erb*) stellt das Grundlayout dar und wird von jedem Controller als Standard verwendet.

```
01| <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
02|   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
03| <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
04|   <head>
05|     <title>ShoppingList</title>
06|     <meta http-equiv="Content-Type"
07|       content="text/html; charset=utf-8" />
08|   </head>
09|   <body>
10|     <%= yield %>
11|   </body>
12| </html>
```

Listing 1.7: app/views/layouts/application.html.erb

- **Zeile 10:** Hier fügt Rails das entsprechende Template der Action in das Grundlayout ein.

Rufen wir nun erneut die Adresse *http://localhost:3000/impressum/index* im Webbrowser auf und betrachten den HTML-Quelltext, bekommen wir ein valides HTML-Dokument angezeigt.

1.6.7 Das Gerüst der Applikation „ShoppingList“ erstellen (Scaffold)

Der Quelltextgenerator scaffold von Rails erstellt uns ein vollständiges CRUD-Gerüst (Create, Read, Update, Delete) für eine bestimmte Ressource. Der Generator legt uns im Wesentlichen folgende Dateien an: die Model-Klasse der Ressource inklusive Migrationsskript für die Datenbank, die Controller-Datei, die View-Template- und Helper-Dateien, ein CSS-Stylesheet und Testdateien für Unit-Tests und Functional Tests. Außerdem wird der neue Controller im Router eingetragen. Mithilfe des Scaffold-Generators lassen sich sehr schnell Applikationsprototypen entwickeln. Für alle Leser, die noch keine Erfahrung mit Rails haben, ist dies eine gute Möglichkeit, um das Zusammenspiel zwischen den verschiedenen Komponenten von Rails besser zu verstehen.

Wir wollen nun mithilfe des Scaffold-Generators ein CRUD-Gerüst für die einzelnen Positionen der Einkaufsliste erzeugen. Der Scaffold-Generator erwartet als ersten Parameter den Namen der Ressource: *rails generate scaffold NAME [field:type field:type ...] [options]*.

```
01| $> rails generate scaffold item what:string how_much:string
02|       invoke  active_record
03|       create    db/migrate/20100101193102_create_items.rb
04|       create    app/models/item.rb
05|       invoke    test_unit
06|       create      test/unit/item_test.rb
07|       create      test/fixtures/items.yml
08|        route  resources :items
09|       invoke  scaffold_controller
10|       create    app/controllers/items_controller.rb
11|       invoke    erb
12|       create      app/views/items
13|       create      app/views/items/index.html.erb
14|       create      app/views/items/edit.html.erb
15|       create      app/views/items/show.html.erb
16|       create      app/views/items/new.html.erb
17|       create      app/views/items/_form.html.erb
18|       create      app/views/layouts/items.html.erb
19|       invoke    test_unit
20|       create      test/functional/items_controller_test.rb
21|       invoke    helper
22|       create      app/helpers/items_helper.rb
23|       invoke      test_unit
24|       create        test/unit/helpers/items_helper_test.rb
25|       invoke  stylesheets
26|    identical    public/stylesheets/scaffold.css
```

- **Zeile 01:** Hier generieren wir das CRUD-Gerüst für die Ressource *Item* mit den beiden Attributen *what* und *how_much*, die beide vom Typ *String* sind.
- **Zeile 02-26:** Hier können wir sehen, welche Verzeichnisse und Dateien vom Scaffold-Generator angelegt wurden.

Werfen wir nun einen Blick auf die erstellte Model-Klasse sowie die entsprechende Datenbankmigration. Die Model-Klasse *Item* enthält nur die Klassendeklaration, da ActiveRecord sich per Konvention die Getter- und Setter-Methoden für die Attribute *what* und *how_much* aus dem Datenbankschema herleitet. Weiteres zu diesem Thema erfahren Sie im Kapitel „ORM-Bibliotheken (Model-Komponente)“.

```
01| class Item < ActiveRecord::Base
02| end
```

Listing 1.8: app/models/item.rb

Ein Migrationsskript dient dazu, Änderungen an der Struktur oder dem Inhalt der Datenbank durchzuführen. In unserem Fall legen wir eine Tabelle *items* mit den beiden Spalten *what* und *how_much* vom Ruby-Typ *String* an. Die Datenbankmigrationen können wir anschließend mithilfe eines Rake Tasks von der Kommandozeile ausführen.

```
01| class CreateItems < ActiveRecord::Migration
02|   def self.up
03|     create_table :items do |t|
04|       t.string :what
05|       t.string :how_much
06|       t.timestamps
07|     end
08|   end
09|
10|   def self.down
11|     drop_table :items
12|   end
13| end
```

Listing 1.9: db/migrate/20100101193102_create_items.rb

- **Zeile 02-08:** Die Klassenmethode *up* wird beim Aufruf des Rake Tasks *rake db:migrate ausgeführt*.
- **Zeile 10-12:** Datenbankmigrationen sollten immer wieder rückgängig gemacht werden können. Hierfür ist die Klassenmethode *down* zuständig, die beim Aufruf des Rake Tasks *rake db:migrate:down* ausgeführt wird.

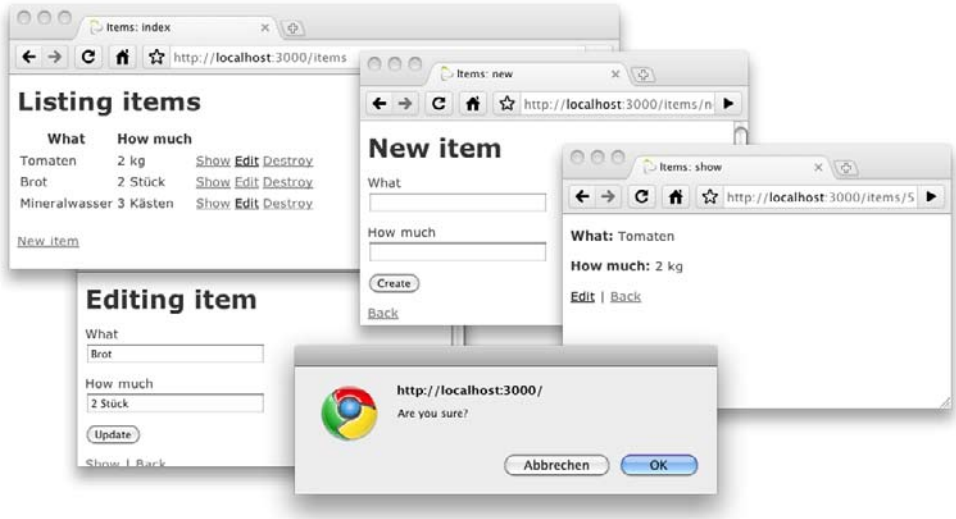
Lassen Sie uns nun die benötigten Datenbanken erstellen und im Anschluss daran die Datenbankmigrationen für das Anlegen der Datenbanktabelle *items* ausführen. Über den Rake Task *rake db:create:all* legen wir alle Datenbanken so an, wie sie in der Datenbankkonfigurationsdatei (*config/database.yml*) definiert sind. Voraussetzung hierfür ist, dass Sie die Konfiguration der Datenbanken bereits vorgenommen haben, wie im Abschnitt „Konfiguration“ beschrieben.

```
01| $> rake db:create:all
```

Nach dem Anlegen der Datenbank können wir nun das Datenbankmigrationsskript für das Anlegen der Datenbanktabelle *items* mithilfe des Rake Tasks *rake db:migrate* ausführen.

```
01| $> rake db:migrate
02| ==  CreateItems: migrating ============
03| -- create_table(:items)
04|    -> 0.0513s
05| ==  CreateItems: migrated (0.0521s) ======
```

Nachdem wir unsere Datenbanktabellen erfolgreich angelegt haben, müssen wir lediglich den Webserver der Applikation starten (*rails server*) und die Adresse *http://localhost:3000/items* im Webbrowser aufrufen. Uns wird dann eine einfache Einkaufsliste angezeigt, bei der wir Positionen hinzufügen, aktualisieren und wieder entfernen können.



Aktuell müssen wir die vollständige Adresse eingeben, um im Webbrowser zu der Einkaufsliste zu gelangen. Wenn wir die Einkaufsliste über die Root-Adresse (*http://localhost:3000*) erreichen wollen, müssen wir hierfür das Routing der Applikation anpassen und die Datei *public/index.html* löschen.

```
01| ShoppingList::Application.routes.draw do |map|
02|   # ...
03|   root :to => "items"
04| end
```

Listing 1.10: config/routes.rb

- **Zeile 03:** Hiermit drücken wir aus, dass der Root-Pfad auf die Ressource *items* zeigen soll.

Ein Standard-CRUD-Controller beinhaltet sieben Methoden. Die Methode *index* liefert uns eine Liste aller Einträge zurück. Die Methode *show* zeigt uns einen einzelnen Eintrag an, die Methode *destroy* dient zum Löschen eines Eintrages. Mithilfe der Methoden *new* und *create* können neue Einträge erstellt werden. Das Aktualisieren eines Eintrags kann über die Methoden *edit* und *update* realisiert werden. Im Kapitel „Programmlogik (Controller-Komponente)“ gehen wir detaillierter auf die Funktionsweise von Controllern ein.

1.6.8 Validierungen hinzufügen

Das Webframework Rails bietet für die meisten Standardaufgaben bereits fertige Lösungen an. Aktuell ist es möglich, bei der Einkaufslisten-Applikation leere Positionen anzulegen, da keines der Eingabefelder ein Pflichtfeld ist. Mithilfe der Validierungsmethoden, die wir in ActiveRecord zur Verfügung haben, können wir ausdrücken, dass ein Model-Attribut ein Pflichtfeld sein soll. ActiveRecord bietet uns eine Vielzahl von Validierungsmethoden an. In unserem konkreten Beispiel benötigen wir die Validierungsmethode *validates_presence_of* um auszudrücken, dass es sich beim Attribut *what* um ein Pflichtfeld handelt.

```
01| class Item < ActiveRecord::Base
validates :what, :presence => true
03| end
```

Listing 1.11: app/models/item.rb

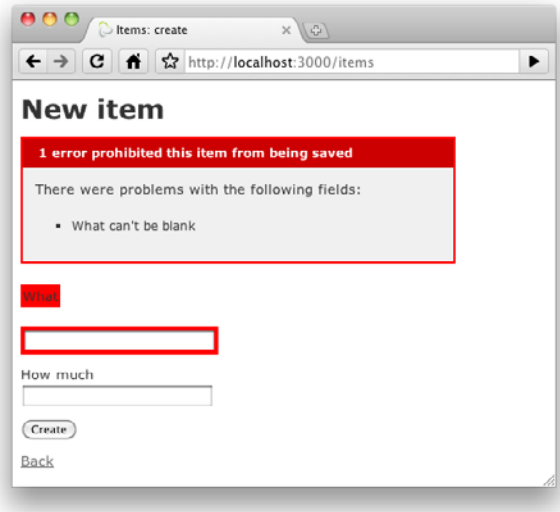
- **Zeile 02:** Hiermit drücken wir aus, dass das Attribut *what* eine Pflichteingabe ist.

Wenn wir das Rails-Plug-in *dynamic_form* installieren (*rails Plug-in install git://github.com/rails/dynamic_form.git*), können wir eine sehr praktische Fehlermeldung generieren lassen, falls Validierungsfehler auftreten.

```
01| <% form_for(@item) do |f| %>
02|   <%= f.error_messages %>
03|   ...
04| <% end %>
```

Listing 1.12: app/views/items/_form.html.erb

Füllen wir nun beim Anlegen einer neuen Position auf der Einkaufsliste das Feld „What“ nicht aus, bekommen wir im Webbrowser nach dem Absenden des Formulars folgenden Hinweis angezeigt:



1.6.9 Automatisiert Testen

Rails veränderte die Webentwicklung mit Blick auf automatisierte Tests enorm und brachte vielen Entwicklern Herangehensweisen an das Testen von Software wie TDD („Test-driven Development“) näher. Rails beinhaltet ein gut integriertes Testframework. Da das Schreiben von sinnvollen Tests ein wichtiges Thema ist, werden wir auf diesen Aspekt im Kapitel „Testen“ tiefer eingehen. In diesem Abschnitt wollen wir mit Ihnen zwei einfache funktionale Tests schreiben.

Der erste Test soll sicherstellen, dass das Impressum erreichbar ist. Hierfür rufen wir die Action *index* des Controllers ImpressumController mittels einer HTTP-Get-Anfrage auf und überprüfen, ob die Antwort den HTTP-Statuscode 200 enthält.

```
01| require File.expand_path(File.dirname(__FILE__)+'/../test_helper')
02|
03| class ImpressumControllerTest < ActionController::TestCase
04|   test "should get index" do
05|     get :index
06|     assert_response :success
07|   end
08| end
```

Listing 1.13: test/functional/impressum_controller_test.rb

- **Zeile 04:** Hier geben wir dem Test einen Namen und drücken aus, was wir mit dem Test testen wollen.
- **Zeile 05:** Hier schicken wir eine HTTP-Anfrage an die Action *index* des Controllers ImpressumController.
- **Zeile 06:** Hier findet unsere eigentliche Überprüfung statt, da wir hier die Behauptung aufstellen, dass die Antwort der HTTP-Anfrage vom Typ *:success* ist, welches dem HTTP-Statuscode 200 entspricht.

Mit dem zweiten Test wollen wir das Hinzufügen einer neuen Position zur Einkaufsliste testen. Hierfür müssen wir überprüfen, ob eine weitere Position in der Datenbank abgespeichert wurde. Rails bietet uns die Test-Helper-Methode *assert_difference*, mit der wir eine Zunahme der Einträge in einer bestimmten Datenbanktabelle überprüfen können.

```
01| require File.expand_path(File.dirname(__FILE__)+'/../test_helper')
02|
03| class ItemsControllerTest < ActionController::TestCase
04|   test "should create item" do
05|     assert_difference('Item.count') do
06|       post :create, :item => { :what => "Tomaten" }
07|     end
08|     assert_redirected_to item_path(assigns(:item))
09|   end
10| end
```

Listing 1.14: test/functional/impressum_controller_test.rb

- **Zeile 05-07:** Hier überprüfen wir das Anlegen einer Position. Dafür senden wir eine HTTP-Post-Anfrage mit dem Parameter *:what* und dem Wert „Tomate".
- **Zeile 08:** Hier überprüfen wir, ob der Benutzer nach dem Anlegen einer Position auf die Action *Show* weitergeleitet wird.

Beim Schreiben von Tests ist es immer wichtig, auch einen Negativtest zu schreiben. Wenn wir die Validierungen in der Model-Klasse *Event* löschen würden, dann liefen unsere Tests aktuell immer noch durch.

```
01| test "can not create a empty position" do
02|   assert_no_difference('Item.count') do
03|     post :create, :item => { :what => "" }
04|   end
05| end
```

Dieser Test stellt nun sicher, dass wir der Einkaufsliste keine leeren Positionen hinzufügen können.

Nachdem wir nun drei Tests geschrieben haben, sollten wir diese abschließend auch ausführen, um zu überprüfen, ob unsere Annahmen auch wirklich stimmen. Über den Rake Task *rake test* führen wir alle Tests der Applikation aus.

```
01| $> rake test
02| Loaded suite ...
03| Started
04| .........
05| Finished in 1.577031 seconds.
06|
07| 9 tests, 12 assertions, 0 failures, 0 errors
```

Die Ausgabe zählt hier mehr Tests als nur unsere drei aus diesem Abschnitt, da weitere Tests von anderen Quelltextgeneratoren angelegt wurden. Möchten wir hingegen eine spezifische Testdatei ausführen, können wir dabei wie im folgenden Beispiel vorgehen.

```
01| $> ruby test/functional/impressum_controller_test.rb
02| Loaded suite test/functional/impressum_controller_test
03| Started
04| .
05| Finished in 0.163179 seconds.
06|
07| 1 tests, 1 assertions, 0 failures, 0 errors
```

Zu beachten ist hierbei, dass die Testdatenbank das aktuelle Datenbankschema beinhalten muss. Dies können wir über den Rake Task *rake db:test:prepare* sicherstellen. Dieser Rake Task wird beim Aufrufen der vollständigen Testsuite (*rake test*) automatisch ausgeführt.

1.6.10 Welche Funktionen bietet uns das Skript „rails“?

Bisher haben wir die Skripte für das Aufrufen der verschiedenen Quelltextgeneratoren und das Skript *rails server* für das Starten des Webservers kennen gelernt. Rails bietet uns allerdings noch einige weitere nützliche Skripte, die wir im Anschluss kurz anschneiden wollen.

rails runner

Mit dem Skript *rails runner* haben wir die Möglichkeit, Ruby-Quelltext im Kontext der Rails-Umgebung auszuführen.

```
01| $> rails runner 'Item.delete_all'
```

rails console

Mit dem Skript *rails console* können wir Ruby-Quelltext in einer interaktiven Shell ausführen. Der Unterschied zu einer normalen IRB-Konsole (Interactive Ruby) liegt darin, dass wir bei der interaktiven Rails-Konsole die Rails-Umgebung zur Verfügung haben und somit ebenfalls auf die ORM-Klassen zugreifen können. Dies erweist sich als äußerst hilfreich, wenn wir bestimmte Ideen schnell mal auf der Konsole ausprobieren möchten.

```
01| $> rails console
02| Loading development environment (Rails 3.0)
03| irb(main):002:0> Item.count
04| => 0
05| irb(main):001:0> Item.create(:what => 'Brot')
06| irb(main):002:0> Item.count
07| => 1
```

rails dbconsole

Mit dem Skript *rails dbconsole* starten wir die Datenbankkonsole. Die Zugangsdaten für die Verbindung zum Datenbankserver werden aus der Datei *config/database.yml* ausgelesen.

```
01| $> rails dbconsole
02| Reading table information for completion of table and column names
03| You can turn off this feature to get a quicker startup with -A
04|
05| Welcome to the MySQL monitor.  Commands end with ; or \g.
06| Your MySQL connection id is 418
07| Server version: 5.0.86 MySQL Community Server (GPL)
08|
09| Type 'help;' for help. Type '\c' to clear
10|
11| mysql> SELECT * FROM items;
12| +-------+------+----------+---------------------+
13| | id    | what | how_much | created_at          |
14| +-------+------+----------+---------------------+
15| | 11010 | Brot | NULL     | 2010-01-05 23:31:34 |
16| +-------+------+----------+---------------------+
17| 1 row in set (0.00 sec)
```

rails plugin

Mit dem Skript *rails plugin* können wir Plug-ins verwalten. Wir haben die Möglichkeit über *rails plugin install* ein Plug-in hinzuzufügen oder über *rails plugin remove* ein Plug-in wieder zu löschen.

```
01| rails plugin install /
02| git://github.com/rails/exception_notification.git
```

rails benchmarker & rails profiler

Mit dem Skript *rails benchmarker* und *rails profiler* können wir verschiedene Performanceanalysen durchführen. Mit dem Skript *benchmarker* können wir zwei verschiedene Implementierungen gegenüberstellen und somit ermitteln, welche der beiden Varianten leistungsfähiger ist.

```
01| rails benchmarker 1000 'Item.all.size' 'Item.count'
02|             user     system      total        real
03| #1    1545.640000  15.750000 1561.390000 (1653.603288)
04| #2      0.790000   0.050000   0.840000 (  5.622144)
```

- **Zeile 01:** Hier vergleichen wir zwei verschiedene Implementierungen, die jeweils 1 000-mal ausgeführt werden.
- **Zeile 02-04:** Hier bekommen wir die Auswertung angezeigt und können den deutlichen Laufzeitunterschied der beiden Implementierungen für das Ermitteln der Anzahl der Einträge in der Datenbanktabelle *items* erkennen.

Mithilfe des Skripts *rails profiler* ist es möglich, genau zu sehen, wo in Ruby wie viel Zeit verbraucht wird. Für die Auswertung des Resultats ist ein recht tiefes Verständnis von Rails sowie Ruby vonnöten. Trotzdem handelt es sich hierbei um ein sinnvolles Tool, um die interne Verarbeitung von verschiedenen Methoden besser zu verstehen.

```
01| $> rails profiler 'Item.count'
02| Using the standard Ruby profiler.
03|   %   cumulative   self              self     total
04|  time   seconds   seconds    calls  ms/call  ms/call  name
05|  22.92     0.66      0.66      115     5.74     9.04  Array#select
06|  18.06     1.18      0.52       39    13.33    19.49  Array#map
07|  12.85     1.55      0.37     7216     0.05     0.05  Hash#key?
08|   8.33     1.79      0.24     7226     0.03     0.03  String#to_s
09| ...
```

1.7 ... und los gehts!

Dieses Kapitel sollte Ihnen einen ersten Einblick in die Thematik Rails und dessen geschichtlichen Hintergrund verschafft haben. Wir hoffen, dass wir Ihnen durch die Entwicklung dieser ersten Rails-Applikation einen Vorgeschmack auf die bevorstehenden Herausforderungen sowie den damit einhergehenden Spaß vermitteln konnten. In den weiterführenden Kapiteln wollen wir nun tiefgründiger auf die einzelnen Thematiken eingehen und Ihnen das entsprechende Know-how näher bringen. Die nachfolgenden Entwicklungsbeispiele werden anhand der bereits bestehenden Veranstaltungsdatenbank für Ausdauerwettkämpfe Kraeftemessen.com beschrieben.