Ruby on Rails/Print version

From Wikibooks, the open-content textbooks collection

< Ruby on Rails

This is the print version of Ruby on Rails
You won't see this message or any elements not part of the book's content when you print or preview this page.

Note: the current version of this book can be found at http://en.wikibooks.org/wiki/Ruby_on_Rails

Contents [hide]

navigation
- Main Page
- Help
- Browse
- Cookbook
- Wikijunior
- Featured books
- Recent changes
- Donations
- Random book

search

Go Search

community
- Reading room
- Community portal
- Bulletin Board
- Help out!
- Policies and guidelines
- Contact us

print/export
- Create a book
- Download as PDF
- Printable version

toolbox
- Search this book
- What links here
- Related changes
- Upload file
- Special pages
- Permanent link

Introduction

Index Next: Getting Started/Install on Windows

Ruby on Rails, or often seen as RoR or just Rails, is an web application framework written in the programming language Ruby. Rails was created to be a project management tool for "37signals". Quickly, it became one of the most popular frameworks on the web and its ideas and philosophies has inspired many frameworks in different languages.

Features

- Rails supports agile software development
- Rails has a short and very intuitive language
- Single blocks of code are intended to be short und clear
- Rails is very flexible
- Rails comes with a testing framework
- The framework follows the principles of DRY (Don't repeat yourself)
- Rails has some conventions: these allow quick development and consistent code
- Rails uses the MVC-Architecture (Model-View-Controller)
- Rails comes with an integrated server for local testing
- Applications with Rails are RESTful

Getting Started

Installation

Previous: Introduction Index Next: Getting Started/Install on OS X

Installation on Windows

To start, you will need the following components:

- Ruby
- RubyGems
- Rails
- a driver for your database

Ruby

Install Ruby as a regular application. You might need administrator privileges to install it successfully. Check the Ruby site for the latest version. The Ruby sites provides packages for all OS. Just follow their steps: ruby website

Gems

RubyGems is a packaged Ruby application. Using the `gem` command helps you installing and deleting gem-packages. Gems allows you to install additional features for your application and easy management of already installed ones.

Download the latest RubyGem package from the offical ruby website. To install, open up a console and navigate to the folder containing the gem file and the file `setup.rb`. To install the gems, type:

```
ruby setup.rb
```

To verify the installation, check the version of gems:

```
gem -v
```

It should display the proper version (1.3.1 as of the writing of this book)

Rails

To install Rails, we can use our freshly installed gems (Rails is a gem). Use the console to type

```
gem install rails
```

This downloads and install all the needed components on your system. After the installation is done, verify the installation by checking if you have the latest version:

```
rails -v
```

It should display the current Rails version (2.3.2 as of writing this guide)

DB Driver

Rails supports a broad range of databases. The default database is SQLite3. You can specify the database you want to use when you first create your Rails project. But no worries, it can be altered any time. For this Wikibook, we will use SQLite3 as database.

To use SQLite3 on your system together with Ruby on Rails, download the latest dll files from the sqlite website. Choose the files that contains the dlls without TCL binding sqlitedll-3 x x.zip. After downloading, copy all the files inside the zip into your Ruby `/bin` directory

At last, we need to install a gem for our database. Since we are using SQLite3 in this book, we want to install the proper gem:

```
gem install sqlite3-ruby --version 1.2.3
```

Even though 1.2.3 is not the current version, it will work as intended because newer version won't work in Windows

Database Configuration

If your first application is created, take a look inside the `/config` folder. We need to tell Rails the name of our database and how to use it. Depending on your chosen database during the creation of the Rails application, the `database.yml` file always look different. Because we decided to stay with the default SQLite3 database, the file will look something like this:

```
# SQLite version 3.x
# gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as <tt>test</tt> will be erased and
# re-generated from your development database when you run <tt>rake</tt>.
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

We may now give our database a proper name. Consider giving different names to your "test"- and to your "production"-database. This is especially important when working in a live-environment and you do not want bad data inside your database.

To create your database in our new project environment, we need to run

```
rake db:create
```

Rake is a build in tool, that allows you to run many pre-made programs to ease up development. There is more functionality inside Rake then just creating a database. But one thing at a time. The `db:create` commando creates the database with the given name (inside db.yml). If you're using SQLite3, you will find a *.sqlite3 file in the `/db` folder. This is your database, stored in a single, handy file. For practicing and local development purposes, this is ideal because all your data is in one file that can be quickly read and written.

Previous: Getting Started/Install on Windows | Index | Next: Getting Started/Install on Linux

Install on OS X

Option 1

- Download Ruby and RubyGems
- Install Ruby
- Install RubyGems
- Get to the command-line and type:

```
sudo gem install rails --include-dependencies
```

Option 2

- Download Locomotive
- Install Ruby on Rails software from Locomotive

Option 3

- Use the broken Ruby installation on OSX 10.4 and follow this guide

RMagic

A shellscript has been made that will install everything you need to make RMagic work on your OS X box. Copy the contents, save it as a file and run the file through a terminal (by navigating to the file and type `./name_of_file`.

Previous: Getting Started/Install on OS X Index Next: Getting Started/Don´t repeat yourself

Install on Linux

Install or update Ruby

It is important that you get the right version of the language. Versions 1.8.2 and 1.8.4 are recommended. 1.8.3 has problems, and 1.8.5 (the current stable version) runs fine, but you will not be able to use breakpointer for debugging (a very handy tool). So I recommend 1.8.4.

Get it from the offical Ruby website

Get it from the SVN repository

Debian

For Debian-based systems (Debian, Ubuntu, etc.), your best bet is to type:

```
sudo apt-get install ruby -t '1.8.4'
sudo apt-get install irb rdoc
```

This should install the Ruby language interpreter, the RDoc documentation generator, and irb command-line interpreter.

Install RubyGems

You should be able to obtain RubyGems by going to the Gems Website and clicking the "download" link. Choose the proper archive and install.

Install Rails

Get to the command-line and type:

```
sudo gem install rails --include-dependencies
```

Concepts

Previous: Getting Started/Install on Linux Index Next: Getting Started/Model-View-Controller

Don´t repeat yourself

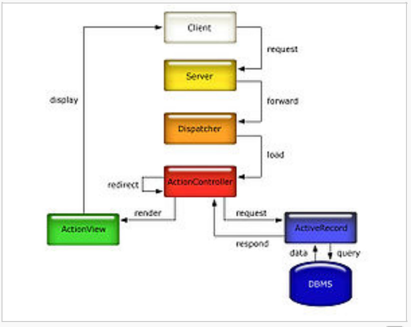
To help to maintain clean code, Rails follows the idea of DRY. The idea behind it, is simple: whenever possible, re-use as much code as possible. This reduces errors, keeps your code clean and even more important: it takes a lot of work off your shoulders by writing code once and using it again. For more information of DRY look at the Wikipedia article

Previous: Getting Started/Don´t repeat yourself Index Next: Getting Started/Convention Over Configuration

Model-View-Controller

As already mentioned, Rails relies on the MVC pattern. Model-View-Controller has some benefits over traditional concepts:

- it keeps your business logic separated from your (HTML-based) views
- keeps your code clean and neat in one place



MVC-Architecture inside Rails

Model

The model represents the information and the data from the database. It is as independent from the database as possible (Rails comes with its own O/R-Mapper, allowing you to change the database that feeds the application but not the application itself). The model also does the validation of the data before it gets into the database. Most of the time you will find a table in the database and an according model in your application.

View

The view is the presentation layer for your application. The view layer is responsible for rendering your models into one or more formats, such as XHTML, XML, or even Javascript. Rails supports arbitrary text rendering and thus all text formats, but also includes explicit support for Javascript and XML. Inside the view you will find (most of the time) HTML with embedded Ruby code. In Rails, views are implemented using *ERb* by default.

Controller

The controller connects the model with the view. In Rails, controllers are implemented as *ActionController* classes. The controller knows how to process the data that comes from the model and how to pass it onto the view. The controller should not include any database related actions (such as modifying data before it gets saved inside the database). This should be handled in the proper model.

Helpers

When you have code that you use frequently in your views or that is too big/messy to put inside of a view, you can define a method for it inside of a helper. All methods defined in the helpers are automatically usable in the views.

Best Practices

Current best practices include:

- fat model and skinny controller
- business logic should always be in the model

- the view should have minimal code
- Use helpers!

Previous: Getting Started/Model-View-Controller Index Next: Getting Started/Creating the application

Convention over Configuration

When starting to work with Rails you will find yourself looking at controllers and lots of views and models for your database. In order to reduce the need of heavy configuration, the team behind Rails has set up rules to ease up working with the application. These rules are not one-way. You may define your own rules but for the beginning (and for your further work, too) it's a good idea to stick to the conventions that Rails offers. These conventions will speed up development, keep your code concise and readable and - most important - these conventions allow you an easy navigation inside your application.

An example should show you how the conventions work together: You have a database table called `orders` with the primary key `id`. The matching model is called `order` and the controller, that handles all the logic is named `orders_controller`. The view is split in different actions: if the controller has a `new` and `edit` action, there is also a new- and edit-view.

First Application

Previous: Getting Started/Convention Over Configuration Index Next: Getting Started/Running the Server

First Application

Create a basic Rails Application

Creating your first Rails application is as simple as typing:

```
rails firstapplication
```

in the console. This will create the Rails application in the current directory. If you need to use another database than the default (SQLite3) you can specify the database with the `-d` parameter. If you need e.g MySQL support, create your app with

```
 rails firstapplication -d mysql
```

NOTE: on Windows, you have to to start the commands with `ruby` explicitly. On Linux and OSX you can just use the command without starting `ruby`.

This sets up the basic files and folders you need. Let's have a look at the files and folders and what they are good for:

The Application Structure

After creating your Rails application, you will be presented with a directory structure. The following table should introduce you to the folders and their purpose:

File/Folder	What is it good for
app/	This is the folder where you will work most of the time. It contains your model, view and the controller.
app/model	Contains all your models for your project.
app/view	Includes all HTML files inside a folder. These files are named after an corresponding action inside your controller. Aditionally, these files are grouped into folders with the controller name.
app/controller	Holds the logic for your Rails application.
db/	All database related files go into this folder. If you are working with SQLite, then the database is likely to be in this folder as *.sqlite3 file.
db/migrate	Has all the migrations that you created.
config/	As the names suggest, it has all the necessary configuration files of your Rails application. There is only very little configuration needed, so no worries that you have to crawl through endless lines of code. Localizations, routes and all basic files can be found here.
public/	All your static files (files that do not change dynamically) your CSS or JavaScript files are inside this folder.
public/images	All your static Images
public/javascripts	When you embedd Javascript, CSS or images, the default location is inside these folders. Rails will automatically look inside these to find the proper file.
public/stylesheets	When you embedd CSS, the default location is inside this folder. Rails will automatically look inside this to find the proper file.
script/	Holds scripts for Rails that provide a variety of tasks. These scripts link to another file where the "real" files are. Inside these folder you will find the generators, server and console.
log/	Log files from your application. If you want to debug some parts of your application you can check out these files.
test/	All files for testing your application.

doc/	Documentation for the current Rails application.
lib/	Extended modules for your application.
vendor/	Whenever you have 3rd-party plug ins, they will be found in this folder.
tmp/	Temporary files
README	Basic guide for others on how to setup your application, what specialities it has or what to be careful about.
Rakefile	Handles all the Rake tasks inside your application.

Basic creation of different files

Most of Rails files can be created via the console by entering a simple command that tells Rails what you want. This way you can create database migrations, controllers, views and much more. All commands look like

```
ruby script/generate generator generator-options
```

To see what options are available, just enter

```
ruby script/generate
```

and Rails will show you all available options and what generators are currently installed. By default, you can choose from the different generators. The most important are:

- controller
- helper
- mailer
- migration
- model
- scaffold

If you want more information on different generators, simply enter the generator command e.g. `script/generate model` in the console and you will get information for this specific command and examples explaining what the generator does.

When working with generators, Rails will create and name all the necessary files in a proper way according to the conventions. This is especially important when working with Migrations (see later). Rails will never overwrite your files if they already exist (unless you tell Rails explicitly to do so).

A very good way to get you started with everything you need is to `scaffold` your files. This will create a basic CRUD-structure for your files. (CRUD=**C**reate **R**ead **U**pdate and **D**elete; this reflects SQL attributes create, insert, update and delete) Scaffolding creates not only your migrations but also a controller with the most basic syntax and a view-folder, that comes with templates for very basic displaying and editing of your data. To use scaffolding, we will use our generators. The scaffolding generator expects the name of a `MODEL/CONTROLLER` and the proper fields and types (in the format field:type).

Say, we want to create tags for different products, we can use:

```
ruby script/generate scaffold Tag name:string popularity:integer
```

Inside the console, Rails will give us information about the files it created:

```
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/tags
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      exists  test/unit/helpers/
      exists  public/stylesheets/
      create  app/views/tags/index.html.erb
      create  app/views/tags/show.html.erb
      create  app/views/tags/new.html.erb
      create  app/views/tags/edit.html.erb
      create  app/views/layouts/tags.html.erb
      identical  public/stylesheets/scaffold.css
      create  app/controllers/tags_controller.rb
      create  test/functional/tags_controller_test.rb
      create  app/helpers/tags_helper.rb
      create  test/unit/helpers/tags_helper_test.rb
      route  map.resources :tags
      dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/tag.rb
      create    test/unit/tag_test.rb
      create    test/fixtures/tags.yml
      exists    db/migrate
      create    db/migrate/20090420180053_create_tags.rb
```

Let's take a quick tour of the files we have now: First Rails created a separate folder inside the view named `tags`. Inside this folder we have some different templatefiles. Besides, we have a controller (`tags_controller.rb`), files for tests (`tags_controller_test.rb, tags_helper_test.rb, tags.yml, tag_test.rb`), helpers (`tags_helper.rb`), a migration file (`20090420180053_create_tags.rb`) and last but not least, a route was also added. As you can see, scaffold does quite a lot of work for us. But remember, that the generated code is very basic and far from being "good" but it already gives you a good starting point.

Previous: Getting Started/Creating the application   Index   Next: Built-In Rails Tools/Generators

Running the server

The bundled Webrick server

As you already know, Rails comes with an integrated server: WEBrick. WEBrick is a Ruby-written server to gets you started right from the beginning. There are alternatives such as Mongrel or Phusion Passenger (formerly known as mod_ruby, a module for Apache). For local(!!) development WEBrick is a good choice.

To start up the server, simply open up a console, navigate to your Rails application and type

- On Windows:

```
ruby script/server
```

- On OS X and Linux:

```
$ script/server
```

After some seconds, WEBrick has been initialized and you are ready to go. The console with the web server needs to stay open, otherwise the server will shut down. To see if everything is working as expected, open up your web browser and switch to

```
http://localhost:3000
```

You should see the default Rails start page saying that everything is working correct. You can view the details page (name) for the current version of your environment and some other variables. The server console not only runs the server, but shows how the requests by the browser are processed, including the amount of queries, the used SQL syntax or the data from your submitted forms.

There are several options, including but not limited to:

- `-p port`: Specify the port to run on
- `-b ip`: Bind to a specific IP address
- `-e name`: Use a specific Rails environment (like `production`)
- `-d`: Run in daemon mode
- `-h`: Display a help message with all command-line options

Mongrel

To start a single mongrel instance:

- On all platforms:

```
mongrel_rails start
```

This should be executed in the root directory of the Rails app you wish to run on Mongrel. There are numerous options you can specify, including:

- `-p port`: run on a specific port
- `-e environment`: execute with a specific Rails environment, like `production`
- `-d`: run in daemon mode

Built-In Rails Tools

Generators

Previous: Getting Started/Running the Server   Index   Next: Built-In_Rails_Tools/Make a generator

Until Make a generator is created, here the link to the next page: Ruby on Rails/Built-In Rails Tools/What is Rake anyway?

Generators

Introduction

Rails comes with a number of generators which are used to create stub files for models, controllers, views, unit tests, migrations and more. Generators are accessed through the command-line script `RAILS_ROOT/script/generate`

All commands look like

```
ruby script/generate generator generator-options
```

To see what options are available, just enter

```
ruby script/generate
```

and Rails will show you all available options and what generators are currently installed. By default, you can choose from the different generators. The most important are:

- controller
- helper
- mailer
- migration
- model
- scaffold

If you want more information on different generators, simply enter the generator commend e.g. "`script/generate model`" in the console and you will get information to this specific command and examples explaining what the generator does.

You can use the generator multiple times for the same controller, but be careful: it will give you the option to overwrite your controller file (to add the actions you specify). As long as you have not modified the controller this might be fine, but if you have already added code then make sure you do not overwrite it and go back and manually add the action methods.

Generate a Model

To generate a model use the following:

```
ruby script/generate model ModelName column:datatype column:datatype [...]
```

Replace ModelName with the CamelCase version of your model name. For example:

```
ruby script/generate model People name:string age:integer
```

This would generate the following:

```
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/people.rb
create  test/unit/people_test.rb
create  test/fixtures/peoples.yml
exists  db/migrate
create  db/migrate/20090607101912_create_peoples.rb
```

Any necessary directories will be automatically created. Existing ones will not be replaced. The file `app/models/people.rb` contains the People class. `test/unit/people_test.rb` contains the unit test stub for the People class. `test/fixtures/peoples.yml` contains a stub for test data which will be used to populate the test database during test runs. The `db/migrate/20090607101912_create_peoples.rb` contains the database migration stub for the Person class. Note the the timestamp at the beginning of the file (`20090607101912`) will always be different depending on the time you created the file.

Generate a Controller

To generate a controller use the following:

```
ruby script/generate controller ControllerName [actions]
```

Replace ControllerName with the CamelCase version of your controller name. When no actions are given, Rails will create a Controller that responds to all 7 REST actions (new, create, update, edit, destroy, index & show)

```
ruby script/generate controller People
```

would generate the following output:

```
exists  app/controllers/
exists  app/helpers/
create  app/views/people
exists  test/functional/
exists  test/unit/helpers/
create  app/controllers/people_controller.rb
create  test/functional/people_controller_test.rb
create  app/helpers/people_helper.rb
create  test/unit/helpers/people_helper_test.rb
```

The file `app/controllers/people_controller.rb` contains the PeopleController class. `test/functional/people_controller_test.rb` contains the functional test stub for the PersonController class. `app/helpers/people_helper.rb` is a stub for helper methods which will be made available to that controller and its associated views. Inside `app/views/people` you will find the created templates for the controller. Depending on the given parameters, there will be different files.

Generate a Migration

To generate a migration use the following:

```
ruby script/generate migration MigrationName column:datatype column:datatype [...]
```

Replace MigrationName with the CamelCase version of your migration name. For example:

```
script/generate migration AddCityToPerson
```

This would generate the following:

```
exists  db/migrate
create  db/migrate/20090607103358_add_city_to_person.rb
```

Migration are automatically generated each time you construct a new model, so you do not need to generate migrations by hand for each model. Typically you will use the migration generator when you need to change an existing model or need join tables. Again, the timestamp starting the filename will be different.Ruby on Rails/Built-In Rails Tools/Make a generator

Rake

Previous: Built-In Rails Tools/Make a generator Index Next: Built-In Rails Tools/Convenient Rake tasks

Rake is a Ruby build tool like make and Ant.

- Rakefiles (rake's version of Makefiles) are completely defined in standard Ruby syntax. No XML files to edit. No quirky Makefile syntax to worry about (is that a tab or a space?)
- Users can specify tasks with prerequisites.
- Rake supports rule patterns to synthesize implicit tasks.
- Flexible FileLists that act like arrays but know about manipulating file names and paths.
- A library of prepackaged tasks to make building rakefiles easier.

Previous: Built-In Rails Tools/What is Rake anyway? Index Next: Built-In Rails Tools/Make your own rake tasks

You can find out what tasks are currently available to Rake with the `rake -T` command. The tasks below are common and used on regular basis.

Database Tasks

- `rake db:migrate [VERSION=x]`: Execute the migrations to the specified version. If the version is omitted then it will migrate to the highest version possible.
- `rake db:sessions:clear`: Clear all of the sessions in the database.

Test Tasks

- `rake test:units`: Execute unit tests

Cache Tasks

- `rake tmp:cache:clear`: Clear out the page/fragment cache.

Previous: Built-In Rails Tools/Convenient Rake tasks Index Next: Built-In Rails Tools/Command-Line Reference

You can make your own rake task by creating a file in the `lib/tasks` directory of your Rails application and adding the Rake tasks to it. For example, adding the following to `lib/tasks/database.rake` will make the `db:recreate` task available to your Rails application:

```
namespace :db do
  desc "Drop and create the current database"
  task :recreate => :environment do
    abcs = ActiveRecord::Base.configurations
    ActiveRecord::Base.establish_connection(abcs[RAILS_ENV])
    ActiveRecord::Base.connection.recreate_database(ActiveRecord::Base.connection.current_database)
  end
end
```

The namespace method puts the contents of the block in the specified namespace. You can nest namespaces as deep as you want although usually one or two nestings is sufficient.

This task can now be executed with

```
rake db:recreate
```

Previous: Built-In Rails Tools/Make your own rake tasks Index Next: ActiveRecord/Naming

Rake can be executed with the following command-line options:

- `--dry-run` or `-n`: Do a dry run without executing actions
- `--help` or `-H`: Display a help message
- `--libdir=LIBDIR` or `-I LIBDIR`: Include LIBDIR in the search path for required modules
- `--rakelibdir=RAKELIBDIR` or `-R RAKELIBDIR`: Auto-import any .rake files in RAKELIBDIR. (default is 'rakelib')
- `--nosearch` or `-N`: Do not search parent directories for the Rakefile
- `--prereqs` or `-P`: Display the tasks and dependencies, then exit
- `--quiet` or `-q`: Do not log messages to standard output
- `--rakefile=FILE` or `-f FILE`: Use FILE as the rakefile
- `--require=MODULE` or `-r MODULE`: Require MODULE before executing rakefile
- `--silent` or `-s`: Like `--quiet`, but also suppresses the 'in directory' announcement
- `--tasks[=PATTERN]` or `-T [PATTERN]`: Display the tasks (matching optional PATTERN) with descriptions, then exit
- `--trace` or `-t`: Turn on invoke/execute tracing, enable full backtrace
- `--usage` or `-h`: Display usage
- `--verbose` or `-v`: Log message to standard output (default)
- `--version` or `-V`: Display the program version
- `--classic-namespace` or `-C`: Put Task and FileTask in the top level namespace

ActiveRecord - The Model

Previous: Built-In Rails Tools/Command-Line Reference Index Next: ActiveRecord/Migrations

Naming

Basics

ActiveRecord uses convention for naming classes, tables and fields. Rails uses convention over configuration. ActiveRecord expects applications to follow certain naming conventions. These conventions extend from file naming, class naming, table naming and more. By default classes are singular, tables are plural, primary keys are `id` and foreign keys are `*table*_id`.

Note: There are also certain names that are reserved and should not be used in your model for attributes outside of the conventions defined in Rails:

- `lock_version`
- `type` - This is only used when you have single table inheritance and must contain a class name
- `id` - Reserved for primary keys
- `table_name_count` - Reserved for counter cache
- `position` - Reserved for acts_as_list
- `parent_id` - Reserved for acts_as_tree
- `lft` - Reserved for acts_as_nested_set
- `rgt` - Reserved for acts_as_nested_set
- `quote` - Method in ActiveRecord::Base which is used to quote SQL
- `template`

Classes

ActiveRecord classes are named in singular form.

Tables

Tables for ActiveRecord objects are named in plural form by default. This pluralization is often an initial point of contention for new Rails users.

If you need to change the table name there are several ways to override the default behavior.

Set use_pluralization

In `config/environment.rb` you can specify `ActiveRecord::Base.use_pluralization = false`. This will apply to all ActiveRecord objects.

Use set_table_name

You can call `set_table_name` to specify a custom table name for a particular model.

For example:

```
  class Dog < ActiveRecord::Base
    set_table_name 'dog'
  end
```

Override table_name

You can also override the `table_name` method and return a value.

For example:

```
  class Dog < ActiveRecord::Base
    def table_name
      'dog'
    end
  end
```

Migrations

[1] Migrations meant to solve the problem of rolling out changes to your database. By defining the changes to your database schema in Ruby files, development teams can ensure that all changes to the database are properly versioned. Additionally migrations help to ensure that rolling out changes to fellow developers as well as other servers (development, QA, production) is handled in a consistent and manageable fashion.

Building a Migration

You can either build the migration on its own using

```
ruby script/generate migration Category
```

and write the specific commands afterwards (if you want to create custom SQL, this is the way to go) or you can create a model that comes with the migration using

```
ruby script/generate model Category name:string amount:integer
```

The console tells you that there were some files created and some already existent. As mentioned before, Rails will never overwrite existing files unless stated otherwise.

Now lets take a look at the migration

```
# 20090409120944_create_categories.rb
class CreateCategories < ActiveRecord::Migration
  def self.up
    create_table :categories do |t|
      t.string :name
      t.integer :amount

      t.timestamps
    end
  end

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

First of all, take a look at the number (*20090409120944*) in front of your file. This is the timestamp of your file and important for the creation of the database tables. This timestamp will always be different, depending on the exact time of the creation of your migration. The idea behind this is to have a "history" of all your migrations available.

But why is this important?

Imagine that you work on a Rails project and you create tables, alter columns or remove columns from your database via migrations. After some time, your client changes his mind and he wants only very basic features and you already started to create advanced features and altered the database. Because you can't remember all the changes that went into the database and their order, you will either spent a lot of time working on the database to have the "old" state available or you have to start from scratch because it would take too long to remember and redo all changes. This is where migration come in handy, because of the timestamp, Rails is able to recognize the changes in their actual order and all changes can be undone easily. Never alter the timestamp manually. This will certainly cause problems. For more on those topic, check out the "managing migrations" section

Speaking of undoing and redoing: notice the two methods inside your migration `self.up` and `self.down`. Both of them do exactly the opposite of each other. While `self.up` creates our categories table with all columns, `self.down` removes (drops) the table from the database with all its contents(!!). When Rails sees that the migration has not been moved to the database, it will use the `self.up` method, if you undo the migration, the `self.down` method gets executed. This way you can make sure that you will always be able to go back to a past state of your database. Keep in mind when writing own migrations always include a `self.up` and a `self.down` method to assure that the database state will be consistent after an rollback.

OK, let's start with the migration itself:

```
create_table :categories do |t|
      t.string :name
      t.integer :amount
      t.timestamps
end
```

We want to create a table called categories(`create_table :categories`) that has a name and an amount column. Additionally Rails adds an timestamp for us where it will store the creation date and the update date for each row. Rails will also create an primary key called **model**_id that auto-increments (1,2,3,...) with every row.

You can choose from a variety of datatypes that go with ActiveRecord. The most common types are:

- string
- text
- integer
- decimal
- timestamp
- references
- boolean

But wait, there is not yet a single table nor column in our database. We need to write the migration file into the database.

```
rake db:migrate
```

handles this job. The command is able to create the table and all the necessary columns inside the table. This command is not limited to migrating a single file so you can migrate an unlimited number of files at once. Rake also knows what migrations are already in the database so it won't overwrite your tables. For more info see "Managing Migrations".

To add a connection between your tables we want to add references in our model. References are comparable to foreign keys (Rails doesn't use foreign keys by default because not all databases can handle foreign keys but you can write custom SQL to make use of foreign keys) and tell your table where to look for further data.

Let's add another model to our already existent database. We want to create a category that has multiply products. So we need to reference this product in our category. We want to create a model:

```
ruby script/generate model Products name:string category:references
```

and insert it into the database

```
rake db:migrate
```

Note the type :references for the category. This tells Rails to create a column inside the database that holds a reference to our category. Inside our database there is now a `category_id` column for our product. (In order to work with these two models, we need to add associations inside our models, see Associations)

Managing Migrations

We already talked about how migrations can help you to organise your database in a very convenient manner. Now we will take a look at how this is achieved. You already know that the timestamp in the filename tells rails when the migration was created and Rake know what migrations are already inside the database.

To restore the state of the database as it was, say 5 migrations before the current, we can use

```
 $rake db:rollback STEP=5
```

This will undo the last 5 migrations that have been committed to the database.

To redo the last 5 steps, we can use a similar command

```
 $ rake db:migrate:redo STEP=5
```

You can also rollback or redo a specific version of a migration state, you just need to provide the timestamp:

```
 $ rake db:migrate:up VERSION=20080906120000
```

Choose whether you want the db_migrate:up method to be executed or the db_migrate:down method

Keep in mind, that restoring your database to a previous state will delete already inserted data completely!

Schema Method Reference

The following methods are available to define your schema changes in Ruby:

- `create_table(name, options)`: Creates a table called name and makes the table object available to a block that adds columns to it, following the same format as `add_column`. See example above. The options hash is for fragments like "DEFAULT CHARSET=UTF-8" that are appended to the create table definition.
- `drop_table(name)`: Drops the table called name.
- `rename_table(old_name, new_name)`: Renames the table called old_name to new_name.
- `add_column(table_name, column_name, type, options)`: Adds a new column to the table called table_name named column_name specified to be one of the following types: :string, :text, :integer, :float, :datetime, :timestamp, :time, :date, :binary, :boolean. A default value can be specified by passing an options hash like { :default => 11 }.
- `rename_column(table_name, column_name, new_column_name)`: Renames a column but keeps the type and content.

- `change_column(table_name, column_name, type, options)`: Changes the column to a different type using the same parameters as `add_column`.
- `remove_column(table_name, column_name)`: Removes the column named column_name from the table called table_name.
- `add_index(table_name, column_names, index_type, index_name)`: Add a new index with the name of the column, or index_name (if specified) on the column(s). Specify an optional index_type (e.g. UNIQUE).
- `remove_index(table_name, index_name)`: Remove the index specified by index_name.

Command Reference

- `rake db:create[:all]`: If :all not specified then create the database defined in config/database.yml for the current RAILS_ENV. If :all is specified then create all of the databases defined in config/database.yml.
- `rake db:fixtures:load`: Load fixtures into the current environment's database. Load specific fixtures using FIXTURES=x,y
- `rake db:migrate [VERSION=n]`: Migrate the database through scripts in db/migrate. Target specific version with VERSION=n
- `rake db:migrate:redo [STEP=n]`: (2.0.2) Revert the database by rolling back "STEP" number of VERSIONS and re-applying migrations.
- `rake db:migrate:reset`: (2.0.2) Drop the database, create it and then re-apply all migrations. The considerations outlined in the note to rake db:create apply.
- `rake db:reset`: Drop and re-create database using db/schema.rb. The considerations outlined in the note to rake db:create apply.
- `rake db:rollback [STEP=N]`: (2.0.2) Revert migration 1 or n STEPs back.
- `rake db:schema:dump`: Create a db/schema.rb file that can be portably used against any DB supported by AR
- `rake db:schema:load`: Load a schema.rb file into the database
- `rake db:sessions:clear`: Clear the sessions table
- `rake db:sessions:create`: Creates a sessions table for use with CGI::Session::ActiveRecordStore
- `rake db:structure:dump`: Dump the database structure to a SQL file
- `rake db:test:clone`: Recreate the test database from the current environment's database schema
- `rake db:test:clone_structure`: Recreate the test databases from the development structure
- `rake db:test:prepare`: Prepare the test database and load the schema
- `rake db:test:purge`: Empty the test database

You can obtain the list of commands at any time using `rake -T` from within your application's root directory.

You can get a fuller description of each task by using

```
rake --describe task:name:and:options
```

See also

The official API for Migrations

References

1. ↑ **Excerpts modified and republished from Steve Eichert's Ruby on rails Migrations Explained article.**

Associations

Introduction

Associations are methods to connect 2 models. Operations on objects become quite simple. The association describes the role of relations that models are having with each other. ActiveRecord associations can be used to describe one-to-one (1:1), one-to-many (1:n) and many-to-many (n:m) relationships between models. There are several types of associations

- *belongs_to* and
- *has_one* form a one-to-one relationship
- *has_one :through* is a different way to create a one-to-one relationship

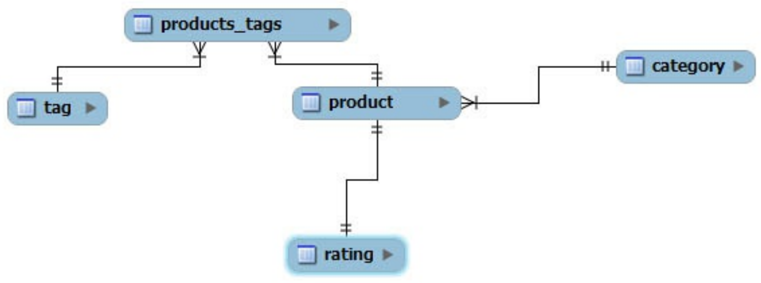
- *has_many* and
- *belongs_to* form a one-to-many relation

- *has_and_belongs_to_many* or an alternative way
- *has_many :through* to create a many-to-many relationship

For all following examples, we assume the following 4 models:

```
class Product < ActiveRecord::Base
end

class Category < ActiveRecord::Base
end

class Rating < ActiveRecord::Base
end

class Tag < ActiveRecord::Base
end
```

Also check out the ER-diagramm for the models below:



belongs_to

The `belongs_to` association sets up an one-to-one (1:1) or one-to-many (1:n) relationship to an other model.

In our example, one Rating belongs to exactly one Product, so we need to set up a `belongs_to` association.

```
class Product < ActiveRecord::Base
  belongs_to :rating
end
```

The product stores an id for a rating, both of them have a one-to-one Relationship. Meaning: One product belongs to one category and one product belongs to one rating.

has_one

The `has_one` assocation is a one-to-one relation too. It declares that each instance of a product from the productmodel possesses only one rating_id.

```
class Rating< ActiveRecord::Base
  has_one :product
end
```

Since the rating and the product form a one-to-one relationship, it is up to us where we put the reference. Because we have put the reference (the rating_id) into the product (see belongs_to :rating into the product model), the association for the rating has to be has_one:product. `Has_one` and `belongs_to` always go together to form a relationship.

has_many

This is a one-to-many connection to an other model. In this case, one product has many categories. Pay attention to the spelling, if you use `has_many`. The model that you want to reference needs to be written in plural.

```
class Category< ActiveRecord::Base
  has_many :products #note the plural here!
end

class Product< ActiveRecord::Base
  belongs_to :category
end
```

The category and the product are forming a one-to-many relationship. Since a category has many products we put the `has_many` association inside the category. **Note** that also the product needs to contain a reference to the category. One product belongs to exactly one category, therefore we put `belongs_to :category` inside the product model.

has_and_belongs_to_many

Also known as : *HABTM*.

`Has_and_belongs_to_many` is the most complex association. You need to keep some things in mind: Because of the way relational databases work, you can not set up a direct relationship. You need to create a joining table. This is easily done with migrations. If we want to create a HABTM association for our product and tag, the association for the joining table might look like following:

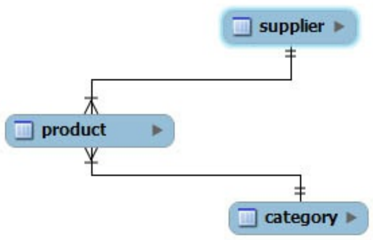
```
class CreateProductTagJoinTable < ActiveRecord::Migration
  def self.up
   create_table :products_tags, :id => false do |t| #we DO NOT need the id here!
     t.integer :product_id #alternatively, we can write t.references :product
     t.integer :tag_id
   end
  end

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

Because we do not need a primary key inside the join table, we use `:id => false` to prevent the automatic creation of a primary key column. The naming of the table needs to be in alphabetical order. "*P*" is before "*T*" in the alphabet so the tablename has to be products_tags (note: plural).

```
class Product< ActiveRecord::Base
  has_and_belongs_to_many :tags
end

class Tag< ActiveRecord::Base
   has_and_belongs_to_many :products
end
```

has_many :through

This association is an other way to form a many-to-many relation. Consider the following ER-Diagramm to show the relationship.



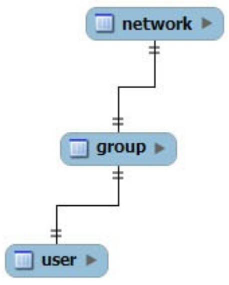
```
class Product < ActiveRecord::Base
  belongs_to :category
  belongs_to :supplier
end

class Supplier < ActiveRecord::Base
  has_many :products
  has_many :categories, :through => :products
end

class Category< ActiveRecord::Base
  has_many :products
  has_many :suppliers, :through => :products
end
```

Instead of using a join table we use another model for the relationship. If we want to find out which supplier is responsible for a specific category we need the product to link the models together. This way we can store additional data inside a real model that also acts as a joining model. Whenever you need to store data that additionally belongs to association (e.g. the date of the creation) you might want to use this type of association.

has_one :through

As with `has_many :through`, `the has_one :through` association uses an "extra" model to form a relation.

Consider this visual relationship:



```
class Network < ActiveRecord::Base
  has_one :group
  has_one :user, :through =>:group
end

class Group < ActiveRecord::Base
  belongs_to :network
  has_one :user
end

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

Note: This example assumes that the membership inside a group is exlusive for each user.

Callbacks

Callbacks provide a means of hooking into an ActiveRecord object's lifecycle.

Implementing Callbacks

There are four types of callbacks accepted by the callback macros:

- Method references (symbol)
- Callback objects
- Inline methods (using a proc)
- Inline eval methods (using a string) - deprecated.

Method references and callback objects are the recommended approaches, inline methods using a proc are sometimes appropriate (such as for creating mix-ins) and inline eval methods are deprecated.

Method Reference

The method reference callbacks work by specifying a protected or private method available in the object, like this:

```
  class Topic < ActiveRecord::Base
    before_destroy :delete_parents

    private
      def delete_parents
        self.class.delete_all "parent_id = #{id}"
      end
  end
```

Callback Objects

The callback objects have methods named after the callback, called with the record as the only parameter such as:

```
  class BankAccount < ActiveRecord::Base
    before_save      EncryptionWrapper.new("credit_card_number")
    after_save       EncryptionWrapper.new("credit_card_number")
    after_initialize EncryptionWrapper.new("credit_card_number")
  end

  class EncryptionWrapper
    def initialize(attribute)
      @attribute = attribute
    end

    def before_save(record)
      record.credit_card_number = encrypt(record.credit_card_number)
    end

    def after_save(record)
      record.credit_card_number = decrypt(record.credit_card_number)
    end

    alias_method :after_find, :after_save

    private
      def encrypt(value)
        # Secrecy is committed
      end

      def decrypt(value)
        # Secrecy is unveiled
      end
  end
```

So you specify the object you want messaged on a given callback. When that callback is triggered the object has a method by the name of the callback messaged.

Proc

Example of using a Proc for a callback:

```
  class Person
    before_save Proc.new { |model| model.do_something }
  end
```

Inline Eval

The callback macros usually accept a symbol for the method they're supposed to run, but you can also pass a "method string" which will then be evaluated within the binding of the callback. Example:

```
  class Topic < ActiveRecord::Base
    before_destroy 'self.class.delete_all "parent_id = #{id}"'
  end
```

Notice that single plings (') are used so the #{id} part isn't evaluated until the callback is triggered. Also note that these inline callbacks can be stacked just like the regular ones:

```
  class Topic < ActiveRecord::Base
    before_destroy 'self.class.delete_all "parent_id = #{id}"',
                   'puts "Evaluated after parents are destroyed"'
  end
```

Callback Reference

before_save

This method is called before an ActiveRecord object is saved.

after_save

before_create

called before creating a new object of the model

after_create

before_update

after_update

before_validation

after_validation

before_validation_on_create

after_validation_on_create

before_validation_on_update

after_validation_on_update

before_destroy

after_destroy

Partially Documented Callbacks

The following callbacks are partially documented. Their use is discouraged because of [1] performance issues.

after_find

The after_find callback is only executed if there is an explicit implementation in the model class. It will be called for each object returned from a find, and thus can potentially affect performance as noted in the [1] Rails API Documentation.

after_initialize

The after_initialize method is only executed if there is an explicit implementation in the model class. It will be called whenever an ActiveRecord model is initialized. It will be called for each object returned from a find, and thus can potentially affect performance as noted in the [1] Rails API documentation.

References

1. ↑ *a b c* "Because after_find and after_initialize are called for each object found and instantiated by a finder, such as Base.find(:all), we've had to implement a simple performance constraint (50% more speed on a simple test case). Unlike all the other callbacks, after_find and after_initialize will only be run if an explicit implementation is defined (def after_find). In that case, all of the callback types will be called."

=

Validations

ActiveRecord supports a variety of model validation methods and allows for addition new methods as needed.

General Usage

For a complete reference of all validations check out the official guide or the API documentation

Additionally you can apply a validation to one or more attributes using the `validate_each` directive:

```
class Person < ActiveRecord::Base
  validates_each :first_name, :last_name do |record, attr, value|
    record.errors.add attr, 'starts with z.' if value[0] == ?z
  end
end
```

It is also possible to define validations via custom methods or blocks:

- validate
- validate_on_create
- validate_on_update

For example:

```
  class Person < ActiveRecord::Base
    validate :validate_email

    def validate_email
      record.errors.add :email, 'Invalid email' unless email =~ /@/
    end
  end
```

Normally validation occurs whenever a record is created or saved, however you can force validation to *not* occur using the `save_with_validation` method passing @false@ as the argument.

Important Validators

validates_acceptance_of

```
validates_acceptance_of :play_rules
```

If you need to check if the user has set or "accepted" a certain checkbox you can use this validation. In this case, the check box with the HTML attribute "name='play_rules'" needs to be checked in order to pass validation.

validates_confirmation_of

This validator checks if an input field has been entered correct both times. For example if you want the user to enter his password 2 times to make sure he enters the correct password (this is seen very often when registering for a site) this is the helper you want to use. To use it, make sure to define it correctly in your view (Note the **_confirmation**):

```
<%= text_field :user, :password%>
<%= text_field :user, :password_confirmation %>
```

validates_format_of

`validates_format_of` accepts a regular expression and checks for the input with the provided pattern given by the :with clause. Also note, that we used a customized message with this helper. This can be done with every validator in the same manner.

```
validates_format_of :username, :with => /\A[a-zA-Z]+\z/,  :message => "Please use only regular letters as username"
```

validates_length_of/validates_size_of

```
validates_length_of :username, :minimum => 5, :maximum => 50
validates_size_of :address, :in => 5..100
```

The `length_of/size_of` validator comes in handy if you need to check the input for a certain length of characters. In the example above, the username should not be longer than 50 characters and not shorter than 5. Alternatively you can specify a range that should be true in order to vaildate. Above the address should constist of 5 to 100 characters. Note that `length_of` and `size_of/` are the same.

validates_numericality_of

```
validates_numericality_of :amount_available
```

Checks if the input is a number of any kind. To make sure, that the input is only an integer, you may use the optional `:only_integer => true` command. There are some useful options with this command for example `:greater_than => 100` makes sure that the given number will be greater then 100: so 101 will be the first valid number.

validates_presence_of

```
validates_presence_of :name
```

One of the most basic validators, this one checks if anything has been inserted in the given form element ("name" in this case).

validates_uniqueness_of

```
validates_uniqueness_of: username
```

Finally, a bit of a more advanced validator: this one checks inside the database if the attribute is already taken or not. In this case, if the user chooses the username "Paul" and the name "Paul" already exists inside the username column of your database, it won't validate.

with scope

It can also validate whether the value of the specified attributes are unique based on multiple scope parameters. For example, making sure that a teacher can only be on the schedule once per semester for a particular class.

```
validates_uniqueness_of :teacher_id, :scope => [:semester_id, :class_id]
```

When the record is created a check is performed to make sure that no record exists in the database with the given value for the specified attribute (that maps to a column). When the record is updated, the same check is made but disregarding the record itself.

Configuration Options

- message - Specifies a custom error message (default is: "has already been taken")
- scope - One or more columns by which to limit the scope of the uniqueness constraint.
- if - Specifies a method, proc or string to call to determine if the validation should occur (e.g. :if => :allow_validation, or :if => Proc.new { |user| user.signup_step > 2 }). The method, proc or string should return or evaluate to a true or false value.

When writing your validation keep in mind that you can mix all of these together and there are even more advanded functions you might want to check out.

Attributes

ActiveRecord attributes are automatically determined from the underlying database schema.

Basic Usage

All of the columns in a table are accessible through methods on the ActiveRecord model. For example:

Consider the following migration:

```
# 20090409120944_create_products.rb
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :name
      t.float :price
      t.integer :amount
    end
  end
#...
end
```

Because Rails has created a primary key for us and called it "id" we can already search for it inside the database. We can interact with the database via the build-in console:

```
 ruby script/console
```

gives us a console that let us communicate with Rails.

Before we can select data from our database it is time to insert some products

```
 >> Product.new(:name => "my book", :price => 14.99, :amount => 4).save
```

if everything went right, Rails will show you a `=>true` inside the console... Let's add some more products

```
 >> Product.new(:name => "my movie", :price => 4.89 :amount => 1).save
```

```
 >> Product.new(:name => "my cd", :price => 9.98, :amount => 12).save
```

Now we have 3 products in our database, going from 1 to 3...

So let's select the first entry

```
 >>Product.find(1)
```

and Rails tells you what it found for this id (remember: find only works when there is a Rails generated id as primary key)

```
 => #<Product: id=1, name="my book" ..... >
```

so we are able to tell Rails to look for our entry with a specific id. But what happens when we have thousands of entries in our database? We can't remember every id of a Product, so Rails offers a very nifty solution:

```
>>Product.find_by_name("my book")
```

```
=> #<Product: id=1, name="my book" ..... >
```

and Rails gives as the output we wanted. This not only works for the name, but for all columns in the database. We could also use `.find_by_price` or `.find_by_amount`

Or you can search for multiple products with their id:

>>Product.find(1,3)

=> #<Product: id=1, name="my book" ..... ><Product: id=3, name="my cd" ..... >

You can also search for all products

```
>>Product.all
```

or

```
>>Products.find(:all)
```

for the first or the last product in your table

```
>>Product.first/last
```

or

```
>>Product.find(:first/:last)
```

If you want more controll over your data, you can use many build-in options or use custom SQL. Let's try to find our data in descending order (3,2,1)

```
>>Product.all(:order => "id DESC")
```

or alternatively

```
>>Product.find(:all, :order => "id DESC")
```

will give you all your prodcuts, starting with the last inserted id - in our case: 3

When we query our database we get lots of infos we are not really interested in such as the "created_at" and "updated_at" column. We only want to take a look at our id and the name so we may use

```
>>Product.all(:select => "id, name")
```

or

```
>>Product.find(:all, :select => "id, name")
```

This method will only display the name and the id of our products making it way more readabel.

When searching for data, keep in mind that you can combine all these methods to your liking. To see more possibilities check out the RoR guide or the API.

Overriding Attributes

You can override any of the accessor methods in your ActiveRecord model classes in order to add logic. For example:

```
class Product
  def name
    self[:name] || 'Unknown'
  end
end
```

```
p = Product.new
```

```
p.name
```

```
=> "Unknown"
```

```
p.name = 'my disk'
```

```
p.name
```

```
=> "my disk"
```

You can access any attribute via `self[attribute_name]` (to get) or `self[attribute_name]=` (to set).

Aggregations

Active Record implements aggregation through a macro-like class method called `composed_of` for representing attributes as value objects. It expresses relationships like "Account [is] composed of Money [among other things]" or "Person [is] composed of [an] address". Each call to the macro adds a description of how the value objects are created from the attributes of the entity object (when the entity is initialized either as a new object or from finding an existing object) and how it can be turned back into attributes (when the entity is saved to the database).

Example:

```
  class Customer < ActiveRecord::Base
    composed_of :balance, :class_name => "Money", :mapping => %w(balance amount)
    composed_of :address, :mapping => [ %w(address_street street), %w(address_city city) ]
  end
```

The customer class now has the following methods to manipulate the value objects:

```
  * Customer#balance, Customer#balance=(money)
  * Customer#address, Customer#address=(address)
```

These methods will operate with value objects like the ones described below:

```
 class Money
   include Comparable
   attr_reader :amount, :currency
   EXCHANGE_RATES = { "USD_TO_DKK" => 6 }

   def initialize(amount, currency = "USD")
     @amount, @currency = amount, currency
   end

   def exchange_to(other_currency)
     exchanged_amount = (amount * EXCHANGE_RATES["#{currency}_TO_#{other_currency}"]).floor
     Money.new(exchanged_amount, other_currency)
   end

   def ==(other_money)
     amount == other_money.amount && currency == other_money.currency
   end

   def <=>(other_money)
     if currency == other_money.currency
       amount <=> amount
     else
       amount <=> other_money.exchange_to(currency).amount
     end
   end
 end

 class Address
   attr_reader :street, :city
   def initialize(street, city)
     @street, @city = street, city
   end

   def close_to?(other_address)
     city == other_address.city
   end

   def ==(other_address)
     city == other_address.city && street == other_address.street
   end
 end
```

Now it's possible to access attributes from the database through the value objects instead. If you choose to name the composition the same as the attributes name, it will be the only way to access that attribute. That's the case with our balance attribute. You interact with the value objects just like you would any other attribute, though:

```
 customer.balance = Money.new(20)     # sets the Money value object and the attribute
 customer.balance                     # => Money value object
 customer.balance.exchanged_to("DKK") # => Money.new(120, "DKK")
 customer.balance > Money.new(10)     # => true
 customer.balance == Money.new(20)    # => true
 customer.balance < Money.new(5)      # => false
```

Value objects can also be composed of multiple attributes such as the case of Address. The order of the mappings will determine the order of the parameters. Example:

```
 customer.address_street = "Hyancintvej"
 customer.address_city   = "Copenhagen"
 customer.address        # => Address.new("Hyancintvej", "Copenhagen")
 customer.address = Address.new("May Street", "Chicago")
 customer.address_street # => "May Street"
 customer.address_city   # => "Chicago"
```

Writing value objects

Value objects are immutable and interchangeable objects that represent a given value such as a Money object representing $5. Two Money objects both representing $5 should be equal (through methods such as == and <=> from Comparable if ranking makes sense). This is unlike entity objects where equality is determined by identity. An entity class such as Customer can easily have two different objects that both have an address on Hyancintvej. Entity identity is determined by object or relational unique identifiers (such as primary keys). Normal `ActiveRecord::Base` classes are entity objects.

It's also important to treat the value objects as immutable. Don't allow the Money object to have its amount changed after creation. Create a new money object with the new value instead. This is exemplified by the Money#exchanged_to method that returns a new value object instead of changing its own values. Active Record won't persist value objects that have been changed through other means than the writer method.

The immutable requirement is enforced by Active Record by freezing any object assigned as a value object. Attempting to change it afterwards will result in a TypeError.

Calculations

Calculations provide methods for calculating aggregate values of columns in ActiveRecord models.

Calculate

All calculations are handled through the calculate method. The calculate method accepts the name of the operation, the column name and any options. The options can be used to customize the query with :conditions, :order, :group, :having and :joins.

The supported calculations are:

- average
- sum
- minimum
- maximum
- count

The calculate method has two modes of working. If the :group option is *not* set then the result will be returned as a single numeric value (fixnum for count, float for average and the column type for everything else). If the :group option is set then the result is returned as an ordered Hash of the values and groups them by the :group column. The :group option takes either a column name or the name of a `belongs_to` association.

Note that if a condition specified in the calculation results in no values returned from the underlying table then the calculate method will return `nil`.

For example:

```
values = Person.maximum(:age, :group => 'last_name')
puts values["Drake"]
=> 43
```

```
drake  = Family.find_by_last_name('Drake')
values = Person.maximum(:age, :group => :family) # Person belongs_to :family
puts values[drake]
=> 43
```

```
values.each do |family, max_age|
   ...
end
```

Average

You can use the average method to calculate the average value for a particular column. For example:

```
Person.average(:age)
```

Will return the average age of all people in the Person model.

An example of customizing the query:

```
Person.average(:age, :conditions => ['age >= ?', 55])
```

This would return the average age of people who are 55 or older.

Sum

The sum method will calculate the sum for a particular column. For example:

```
Product.sum(:number_in_stock)
```

Will return the sum of products in stock.

An example of customizing the query:

```
Product.sum(:number_in_stock, :conditions => ['category_id => ?', 10])
```

Will return the sum of the number of products which are in category 10 and in stock.

Minimum

The minimum method will calculate the minimum value for a particular column. For example:

```
Donation.minimum(:amount)
```

Will return the lowest amount donated.

An example of customizing the query:

```
Donation.minimum(:amount, :conditions => ['created_at > ?', 1.year.ago])
```

This will return the lowest amount donated within the last year.

Maximum

The maximum method will calculate the maximum value for a particular column. For example:

```
Donation.maximum(:amount)
```

Will return the largest amount donated.

An example of customizing the query:

```
Donation.maximum(:amount, :conditions => ['created_at > ?', 1.year.ago])
```

This will return the largest amount donated within the last year.

Count

Counts the number of items matching the condition(s).

```
TestResult.count(:all)
```

Will return the number of TestResult objects in the database.

An example of customizing the query:

```
TestResult.count(:all,:conditions=>['starttime>=?',Time.now-3600*24])
```

Will return the number of TestResult objects whose starttime field is within the last 24 hours.Ruby on Rails/ActiveRecord/Acts as

ActionView - The View

Rendering and Redirecting

Introduction

You already know how to manage your data with ActiveRecord. Now it is time to display your data. All data the view displays comes from the controller. Most of the time, you will work with HTML but you can also use Javascript inside your views (which of course can again be Rails generated) or different CSS.

The "Convention over Configuration" is also an essential part of the view: as mentioned in the beginning of this book, Rails is able to know which file for the view belongs to which action inside in the controller:

```
app/controller/products_controller.rb
```

```
#...
def show
  @product= Product.find(params[:id])
end
#...
```

This action inside our products controller assumes that there is a view responding to the name:

```
app/views/products/show.html.erb
```

As you can see the file has 2 extensions: one is for the browser to display (HTML) and the other one tell Rails how to process it (erb= embedded ruby).

Rendering and Redirecting

You will come across 2 often used methods to display data `render` and `redirect_to` A good example of how these two methods work can be seen in the example below:

This actions gets called when the user submitted updated data

```
def update
    @product= Product.find(params[:id])

    if @product.update_attributes(params[:name])
      redirect_to :action => 'index'
    else
      render :edit
    end
end
```

As you can see both of our methods are used in this simple example. Whenever we successfully updated the name of a product, we get redirected to the index site of the products. If the update fails, we want to return to the edit view.

There is an important difference between `render` and `redirect_to`: render will tell Rails what view it should use (with the same parameters you may have already sent) but `redirect_to` sends a new request to the browser.

Render

Remember the "update" action above? When the update fails, we want to render the edit view with the exact same parameters as we did before, in this case we look for the "id" inside the database and populate the page accordingly. If you want to render another view use

```
render 'categories/show'
```

You can also display a file that is somewhere completely different on your web server

```
render :file => "/u/apps/some_folder/app/views/offers/index"
```

And of course, you can render simple text

```
render :text => "Hello World"
```

You may have already noticed that there is a "layout" folder inside your view. Whenever you use scaffolding to create parts of your application a file inside layout gets created. If you scaffold "Products" the file inside layout will be called `products.html.erb`. This file is responsible for the basic display of your sites matching the common name (in this example, it's products). Whenever you want to redirect your user to another layout you can use

```
render :layout => 'another_layout'
```

Whenever there is no proper layout file, Rails will display the page only with the styling provided inside the requested view. To use a specific layout inside your whole controller you can define the layout inside your Controller

```
class ProductsController < ApplicationController
  layout "my_layout"
  #our actions
end
```

For more infos about layouts, see "Layout Files"

redirect_to

You can use `redirect_to` in a similar manner as you do with render, but keep the big difference between `render` and `redirect_to` in mind.

With `redirect_to` you can easily send the user to a new resource, for example the index page of our products. To learn more about the paths and routing see the chapter on "routing"

```
 redirect_to products_path
```

A very handy `redirect_to` option is `:back`

```
 redirect_to :back
```

will send the user back to the site that he came from

Templates

There are several templating systems included with Rails each designed to solve a different problem.

- ERb - Embedded Ruby is the default templating system for Rails apps. All files ending with `.rhtml` are considered ERb templates.
- Builder - Builder templates are programmatic templates which are useful for rendering markup such as XML. All templates ending with `.rxml` are treated as builder templates and include a variable called `xml` which is an instance of XmlMarkup.

In addition to the built in template systems you can also register new template handlers using the `ActionView::Base.register_template_handler(extension, class)` method. A template handler must implement the initialize(base) method which takes the `ActionView::Base` instance and a `render(text, locals)` method which takes the text to be rendered and the hash of local variables. =

Layout Files

In the previous chapter you learned how to render output to the default layout or to a special layout provided by you. You may have already looked inside such a layout file. Its default content will be similar to

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Products: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<p style="color: green"><%= flash[:notice] %></p>
<%= yield %>
</body>
</html>
```

So you now notice some things that are typical for a layout: the head section has an embedded tag for a stylesheet and the complete content is represented using a yield-tag

Asset tags

All asset tags, such as the "`stylesheet_link_tag`" are little helpers that generate the proper HTML for you. You can use this tags to embed CSS, Java Script and images or to provide RSS-feeds.

To embedded Javascript or CSS, you can use the according helpers:

```
<%= javascript_include_tag "my_javascript" %>
```

or if you want to include CSS

```
<%= stylesheet_link_tag "my_css" %>
```

All these files are located in the public directory `public/javascripts` or `public/stylesheets`. There is no need to provide the extension as Rails will handle it automatically. If you want to include multiple files just provide all the file names inside the tags:

```
<%= stylesheet_link_tag "my_css", "my_other_css" %>
```

Of course these files can be placed anywhere in your application, just be sure to provide the proper path:

```
<%= stylesheet_link_tag "my_css", "files/stylesheet" %>
```

To load all files inside your `public/javascripts` or `public/styleheets` use the appropriate tag with `:all`

```
<%= javascript_include_tag :all %>
```

When embedding CSS you can specify the media attribute inside the tag:

```
<%= stylesheet_link_tag "my_print", media => "print" %>
```

No modern web site is complete without using pictures and graphices so Rails provides you with its own image tag:

```
<%= image_tag "my_image" %>
```

As with JavaScript or CSS the Rails will look into `public/images` by default to find the proper picture. **Note:** that you do not need to tell Rails if you use .jpg, .gif or .png! You can also provide a custom path and HTML attributes

```
<%= image_tag "images/my_image", :height => 50, :width => 50, :alt => "This is my image" %>
```

Yield

Whenever you come across "yield" you will know that this is sort of a placeholder for the view that will be displayed inside your layout, but you are not limited to using a single yield:

```
<body>
  <%= yield :header %>
</body>
```

To get access to the `:header` you can use the "`content_for`" method. So with the above example, that might be:

```
<% content_for :header do %>
  <h1>Hello Rails - This is my header</h1>
<% end %>
```

Named yields can be used for menus, footers or sidebars to ease up maintainance. Just put them in your code and they will be placed in the right place.

Partials

It is time for a little more DRY. If you have a lot of HTML code that repeats itself in many different views (such as forms, dynamic menus,...) you might want to consider using partials: With partials, you just move the code that you use often into a separate file and place a link inside the files that should use your partials

```
<%= render :partial => "form" %>
```

This will look inside the folder from where the partial gets called for a file named `_form.html.erb`. Keep in mind to use an underscore in your filename to tell Rails that it is a partial.

To move your partial to a seperate folder use

```
<%= render :partial => "partials/form" %>
```

will look in "partials" for a file called `_form.html.erb`

Overall partials are a very good way to minimize the effort of copying the same code over and over, but to keep the needed code in single files that can be altered easily and used as often as you want to.

Forms

Basic

Forms are an important part of every web site. We use them to contact the site owners to provide log in credentials or to submit data. Rails has vast support for forms and many build in tags to help you create forms quick and easily.

The most basic form you can get is a simple input field for your name:

```
<% form_tag do %>
  <%= label_tag(:my_name, "My name is:") %>
  <%= text_field_tag(:my_name) %>
  <%= submit_tag("Process") %>
<% end %>
```

This will create a very basic form like

```
<form action="/products/test/1" method="post">
 <div style="margin:0;padding:0">
 <input name="authenticity_token" type="hidden" value="KBBEVzSZOkCi/s9LKhYvW/gdwyZzwH2n39py5FggaT4=" />
 </div>

 <label for="my_name">My name is:</label>
 <input id="my_name" name="my_name" type="text" />
 <input name="commit" type="submit" value="Process" />
 </form>
```

This was achieved by creating a simple action called `test.html.erb` inside our products view,

```
def test
  #nothing here yet
end
```

An empty action inside the controller was added. The page was called inside the browser `http://localhost:3000/products/test/1`. Because we haven't altered the routes we have to provide an id of a product. Additionaly Rails has created a "authenticity_token" to provide security features to our form. You also notice that there is an id for our input. Rails provides the id for every form element with the names you provided. Of course this can be altered by providing the proper option.

The following list should give you an overview of some form tags. As most of the tags work in a similar manner, it should be easy to figure out how much of them works. Also be sure to take a look at the official API.

Radio Boxes:

To create radioboxes simply use

```
<%= radio_button_tag :category, "books", true %>
```

The above will create

```
<input id="category_books" name="category" type="radio" value="books" />
```

If you want to change the id you can pass your own id to to tag `(:id => "someid")`. To preselect an item the 3 option that the helper receives is supposed to be a boolean.

Submit Button:

```
<%= submit_tag ("Save products", :class => "button") %>
```

will create a submit button

```
<input name="commit" type="submit" class="button" value="Save products" />
```

Text fields:

```
<%= text_field_tag ('rails') %>
```

will create an emtpy text field:

```
<input id="rails" name="rails" type="text" />
```

As usual you can add HTML options or specific values

```
<%= text_field_tag ('price', nil, :maxlength => 10) %>
```

notice that we use "`nil`" here. This is because we want to create a text field with no pre-defined value:

```
<input id="price" name="price" type="text" maxlength="10" />
```

If we would have provided a name, Rails would add the HTML attribute "value":

```
<input id="price" name="price" type="text" maxlength="10" value="my predefined value instead of nil" />
```

Select boxes:

A common task inside forms is to let the user select something from select boxes - or drop downs. To display a simple select box we can combine some handy methods:

```
<%= select_tag (:name, options_for_select([['Peter','pete'],['Joseph', 'jo']]))%>
```

would create

```
<select id="name" name="name">
 <option value="pete">Peter</option>
 <option value="jo">Joseph</option>
</select>
```

Another nice helper of the `select_*` family is `select_year`. When you want the user to choose his year of birth, you may use something like:

```
<%= select_year Date.today, :start_year => 2009, :end_year => 1900  %>
```

This will create a select box with all years, starting from 2009 going to 1900 with the current year (`Date.today` as the first argument) being pre-selected. Be sure to check out the other helpers for dates Datehelpers

Handling Errors

Now that you know how to display forms correctly in Rails, we are looking for a way to give the user some feedback over his input. Either if it is correct or if he needs to re-enter some values. You have already read about how to validate form data before it gets written into the database. Now you will see how to display the errors. Basically there are two methods that allow us to display errors: `error_messages` and `error_messages_for`. The name gives a pretty good idea of the difference between these two: `error_messages` displays all errors for the form, while `error_messages_for` displays the errors for a given model.

Using the methods is easy:

```
<% form_for(@product) do |f| %>
  <%= f.error_messages %>
    <p>
    <!-- different form tags -->
    </p>
<% end %>
```

will display all the error messages: either default ones or the ones you have specified inside your model (`:message => "Only integers allowed"`). `error_messages_for` is used in a similar manner. Then it display the error message for the field that matches the given name (`:name` in this case)

```
<%= error_messages_for :name %>
```

You can also customize the error messages that are displayed in a box, even more:

```
<%= f.error_messages :header_message => "Invalid product!", :message => "Correct your entries", :header_tag => :h6 %>
```

Advanced Forms

Until now we mainly worked with form elements that have a `_tag` at the end of the name. When working with a model (e.g. you want to update your data or create a new data set) you want to use the form elements that are more suited for the job. There are no new elements to learn we just use the one we already know but do not add the _tag at the end (compare `form_tag` and `form_for`).

Let's asume we want to create a new Product via a form:

We have a controller that has an action... `products_controller.rb`

```
#...
def new
  @product= Product.new
end
#...
```

... and a view (`view/products/new.html.erb`) that works with our `@product` instance:

```
<% form_for :product, @product, :url=>{:action => "create"} do |f| %>
  <%= f.text_field :name %>
  <%= f.text_field :price %, :size => 10 >
  <%= f.text_field :amount%, :size =>5 >
  <%= submit_tag "New Product" %>
<% end %>
```

will create HTML similar to:

```
<form action="/products" class="new_product" id="new_product" method="post">
 <div style="margin:0;padding:0">
 <input name="authenticity_token" type="hidden" value="bgqa1sbwcC5J/tIpPtJjX/3slveFNJg3WZntSOyHT4g=" />
 </div>
 <input id="product_name" name="post[name]" size="30" type="text" />
 <input id="product_price" name="post[price]" size="10" type="text" />
 <input id="product_amount" name="post[amount]" size="5" type="text" />
 <input id="product_submit" name="commit" type="submit" value="Update" />

</form>
```

Let's inspect the code: `form_for` has some code on his side, `:product` references the name of the model that we want to use and `@product` is the instance of the object itself (in this case, it's going to be a new product). We loop or "yield" through the `form_for` object with the variable *f*. The rest uses code you are already familiar with. We use the action "create" to handle the creation of the file.

Restful Resources

The methods shown above may not be exactly what you will see in other Rails applications. Most of them will use RESTful resources. This gives Rails the option to decide what to do with the data. This makes the `form_for` method a lot easier and more readably:

```
form_for :product, @product, :url=>{:action => "create"}
```

turns into

```
form_for (@product)
```

That is a lot better, isn't it? Now the `form_for` tag looks the same, no matter if you are creating a new product or if you update an existing one. Rails is smart enough to recognize the action and provides us with the correct HTML attributes (see `<form action="/products" class="new_product" id="new_product" method="post">`)

To learn more about routing and REST be sure to take a look at Routing.

Using select boxes with a model

We already learnt how to create select boxes with built in helpers. But you may also want to display the contents of a model inside a select box. To show how this is done, we want to select categories from the database, so the user can select the according one:

```
<%=f.collection_select :category, :id, Category.all, :name, :name.downcase %>
```

If you have some categories inside your database the HTML may look similar to

```
<select id="category_id" name="category[id]">
 <option value="cds">CDs</option>
 <option value="books">Books</option>
 <option value="dvds">DVDs</option>
</select>
```

Notice the `.downcase`: this will write the HTML values lowercase, if your database needs to work with it that way. This is just a starting point for your work. To see more examples and explanations, a good start is the official API

Uploads

The following paragraph should give you just a quick overview on the topic of uploading data with forms. For the more advanced purposes you might want to consider using widely known gems that handle the job. Take a look at the Resources page to find good starting points for your search. Or take a look at the official Rails guide to see a sample action

As with everything you want to upload via a form, you need to set the form to support "`multipart/form-data`". If you want to upload, e.g a picture for our products, you can use these ways: Here, you will need to write an action inside the controller that handles the upload of the file on the server:

```
<% form_tag({:action => :upload}, :multipart => true) do %>
  <%= file_field_tag 'picture' %>
<% end %>
```

For a form bound to a model we can use the already known `form_for tag` (this allows you to save e.g. the image name inside your database)

```
<% form_for @person, :html => {:multipart => true} do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Custom Helpers

Rails comes with a wide variety of standard view helpers. Helpers provide a way of putting commonly used functionality into a method which can be called in the view. Helpers include functionality for rendering URLs, formatting text and numbers, building forms and much more.

Custom Helpers

Custom helpers for your application should be located in the `app/helpers` directory.

Application Helper

The file

```
app/helpers/application.rb
```

contains helpers which are available to all views.

Controller Helpers

By default other helpers are mixed into the views for based on the controller name. For example, if you have a ProjectsController then you would have a corresponding ProjectsHelper in the file

```
app/helpers/projects_helper.rb
```

Example

The following is an example of an Application Helper. The method title will be available to all views in the application. Methods added to this helper will be available to all templates in the application.

```
module ApplicationHelper
    def title
      t = 'My Site'
      t << ": #{@title}" if @title
      t
    end
  end
```

ActionController - The Controller

Introduction

Now that you worked your way though Models and Views, it's time to get to the last part of MVC: the Controller. Your controller creates the proper data for the view and handles some logic structures. In order to fully understand the whole framework, you should also read about "Routing" to fully understand how things work together.

Actions

Actions are methods of your controller which respond to requests. For example:

```
class PeopleController < ApplicationController
      def index
        @people = Person.find(:all)
      end
      def show
        @people= Person.find(params[:id])
      end
    end
```

In this example the *people* controller has two actions: `index` and `show`. The action called `index` is the default action which is executed if no action is specified in the URL. For example:

```
http://localhost:3000/people
```

The `index` action can also be called explicitly:

```
http://localhost:3000/people/index
```

The show action must be called explicitly unless a route has been set up (routing will be covered later):

```
http://localhost:3000/people/show/1
```

In the example above the number 1 will be available in params[:id]. This is because the default route is:

```
map.connect :controller/:action/:id
```

This indicates that the first part of the path is the controller name, the second part the action name and the third part is the ID. A detailed explanation of routes is provided in its own chapter.

Parameters

Remember the "rendering and redirecting" example you've seen when learning about the view?

```
def update
    @product= Product.find(params[:id])

    if @product.update_attributes(params[:name])
      redirect_to :action => 'index'
    else
      render :edit
    end
end
```

You know what the differences between `render` and `redirect_to is`. Now we want to take a look at WHAT gets updates. This is determined by given the "update_attributes" method what we want to update: `params[:name]`. In this example, the data is likely to come from an HTML-Form (therefore a POST request). You can also work with GET requests in the same manner. Rails will handle both requests by using the params command.

When using e.g. check boxes you are likely to get more then one set of data for the same attribute. For example `option[]=1&option[]=2&option[]=3`. As with the example above, you can work with `params[:ids]` to get access to these settings. Your options will look like `options => {'1','2','3'}`

Session

For a technical explanation of a Session take a look at the wikipedia article about Sessions

In Rails you have some options to store the session. Most of the time you want to store the session on the server, but with security-relevant data, you might want to consider storing the session inside a database. To change the session storage, edit `config/initializers/session_store.rb` and be sure to read on the RoR Website carefully.

Work with your session

As with the parameters, Rails provides a simple way of accessing your session. Consider following example:

```
def show_details
  #we may use this inside a user-specific action
  User.find(session[:current_user_id])
end
```

As you can see, you access the session in a similar way to the parameters. Storing a session isn't much more complicated:

```
def index
  #we have some code here to get the user_id of a specific (logged-in) user
  session[:current_user_id] = id
end
```

To destroy the session, just assign it a nil-value

```
session[:current_user_id] = nil
```

Displaying a Flash-message

Flashes are very special and useful part of a session. You may have already found it in one of the view files. Here is how they work: As said, Flashes are special. They exist only once and are destroyed after each request. Flashes are useful to display error messages or notices to the user (e.g. when he tries to log in or if his request resulted in an error)

Inside an action flashes can be used similar to:

```
def check
  #code that does some validation
  flash[:notice] = "Successfull logged in"
end
```

Inside the view you can access it like:

```
<% if flash[:notice] -%>
    <%= flash[:notice] %>
<% end -%>
<!-- maybe some HTML-Code -->
<% if flash[:warning] -%>
    <%= flash[:warning] %>
<% end -%>
```

As you can see from the example above you are not limited to a single flash. You can access multiple flashes by their name you have defined inside the controller.

Cookies

Of course you can work with cookies inside your Rails application. This is, again very similar to working with parameters or sessions. Consider the following example:

```
def index
  #here we want to store the user name, if the user checked the "Remember me" checkbox with HTML attribute name="remember_me"
  #we have already looked up the user and stored its object inside the object-variable "@user"
 if params[:remember_me]
   cookies[:commenter_name] = @user.name
 else # Delete cookie for the commenter's name cookie, if any
  cookies.delete(:commenter_name)
 end
end
```

Routing

Because Routing is such an important part of Rails we dedicated a whole chapter to Routing (even though they are part of ActionView).

For example, if you want to display the product with the id 4, you will use a link similar to products/4. So Rails will figure out, that you want to show the information that belongs to the product with the id 4.

Routing also works when you want to link somewhere from one point of your application to another point. If you want to go back from the products view to the index overview that displays all products, you may place something like this in your view:

```
 <%= link_to 'Back', products_path %>
```

When writing your own routes inside `routes.rb`, keep in mind, the lower the route is inside your file, the less priority it has.

Understanding Routes and Routing

RESTful routes

RESTful routes are the default routes in Rails. To get a more detailed technical view on REST, check out the Wikipedia article.

Basically REST provides a way of communication inside your application and all requests that exist from external sources (just as a browser request). To understand these principles better, take a look the following table:

HTTP Verb	URL	Controller	Action	used for
GET	/products	Product	index	display all products in an overview
GET	/products/new	Product	new	return an HTML form for creating a new product
POST	/products	Product	create	create a new product
GET	/products/1	Product	show	display a specific product
GET	/products/1/edit	Product	edit	return an HTML form for editing a product
PUT	/products/1	Product	update	update a specific product
DELETE	/products/1/	Product	destroy	delete a specific product

As you can see, all the actions of REST are already in our scaffolded controller. Keep in mind that RESTfoul routes reference a single object (products in this case). These 7 actions would result in a single route inside `routes.rb`:

```
map.resources :products
```

With a RESTful resource, it's easy to link the different views together or link to a specific view. With REST, Rails provides us some helpers to get to our desired location:

- products_url & products_path => redirects us to the index overview and edit view for our products (note the plural)
- new_product_url & new_product_path => will lead as to the form that creates a new product
- edit_product_url & edit_photo_path => provides us with an edit-form for a specific product
- product_url & product_path => is responsible for showing, deleting und updating a product

While *_path will create a relative path to, *_url provides the whole URL

- _path => /products
- _url => http://localhost/products

You will very likely need more than one REST route. You can easily write multiple REST routes into a single:

```
map.resources :products, :categories, :customers
```

You will come across many similar constructs like our products-category relation:

```
class Category < ActiveRecord::Base
  has_many :products
end

class Product < ActiveRecord::Base
  belongs_to :categories
end
```

HTTP Verb	URL	Controller	Action	used for
GET	/products/1/categories	Category	index	will show you an overview of the categories for product with the id 1
GET	/products/1/categories/new	Category	new	will give you an HTML form to create a new category for product with id 1
POST	/products/1/categories	Category	create	will create a new category for product with id 1
GET	/products/1/categories/1	Category	show	shows you the categories with id 1 that belongs to your product with id 1

GET	/products/1/categories/1/edit	Category	edit	gives you an HTML form to edit your category with id 1 that belongs to product with id 1
PUT	/products/1/categories/1	Category	update	updates category with id 1 that belongs to product with id 1
DELETE	/products/1/categories/1	Category	destroy	deletes category with id 1 that belongs to product with id 1

As with resources that are not nested, you will be able to access all *_url and *_path helpers e.g. `products_categories_path` or `new_product_category_url`

These path need to be present in your `routes.rb` in an similar maner to your model:

```
map.resources :products, :has_many => :categories
```

if you need to add more than association, put these in []

which is the same as the following route, only shorter

```
map.resources :magazines do |magazine|
  magazine.resources :ads
end
```

With this way, you can nest as many resources as you want, but you should try to keep the level of nesting as low as possible. So one nested resource is ok, but try to avoid 2 or more. To avoid these problems, we can use "shallow nesting"

If we want to add supplier for specific categories, we may end up with something like

```
map.resources :publishers, :shallow => true do |publisher|
  publisher.resources :magazines do |magazine|
    magazine.resources :photos
  end
end
```

or in short:

```
map.resources :products, :has_many => { :categories=> :suppliers }, :shallow => true
```

There are many more advanced features for routing. More infos and instructions can be found in the official Rails guides.

Regular Routes

Even though RESTful routes are the encouraged way to go, you can also use regular routes inside your application. When working with regular routes, you give Rails some keywords and it will map the proper path for you. One of these default routes is already inside your `routes.rb` (this time we don't use `.resources` but `.connect`)

```
map.connect ':controller/:action/:id'
```

will for example be a browser request similar to `products/show/2`

If you are familiar with other languages that focus an the web, you may wonder how query strings inside the URL are handles: Rails automatically provides these parameters inside the params hash

So if you take the example above and add `products/show/2?category=3`, we would be able to access the category id with `params[:category_id]`.

See also

The API with lots of examples

ActiveSupport

Active Support is a collection of various utility classes and standard library extensions that were found useful for Rails. All these additions have hence been collected in this bundle as way to gather all that sugar that makes Ruby sweeter.

ActionMailer

Note: The following documentation was taken directly from the Rails API documentation

ActionMailer allows you to send email from your application using a mailer model and views.

Model

To use ActionMailer, you need to create a mailer model.

```
script/generate mailer Notifier
```

The generated model inherits from `ActionMailer::Base`. Emails are defined by creating methods within the model which are then used to set variables to be used in the mail template to change options on the mail, or to add attachments.

Examples:

```
 class Notifier < ActionMailer::Base
   def signup_notification(recipient)
     recipients recipient.email_address_with_name
     from       "system@example.com"
     subject    "New account information"
```

```
      body        "account" => recipient
    end
  end
```

Mailer methods have the following configuration methods available.

- `recipients`: Takes one or more email addresses. These addresses are where your email will be delivered to. Sets the To: header.
- `subject`: The subject of your email. Sets the Subject: header.
- `from`: Who the email you are sending is from. Sets the From: header.
- `cc`:Takes one or more email addresses. These addresses will receive a carbon copy of your email. Sets the Cc: header.
- `bcc`: Takes one or more email address. These addresses will receive a blind carbon copy of your email. Sets the Bcc: header.
- `sent_on`: The date on which the message was sent. If not set, the header wil be set by the delivery agent.
- `content_type`: Specify the content type of the message. Defaults to text/plain.
- `headers`: Specify additional headers to be set for the message, e.g. headers 'X-Mail-Count' => 107370.

The body method has special behavior. It takes a hash which generates an instance variable named after each key in the hash containing the value that the key points do.

So, for example, body "account" => recipient would result in an instance variable @account with the value of recipient being accessible in the view.

Mailer Views

Like ActionController, each mailer class has a corresponding view directory in which each method of the class looks for a template with its name. To define a template to be used with a mailing, create an .rhtml file with the same name as the method in your mailer model. For example, in the mailer defined above, the template at

```
app/views/notifier/signup_notification.rhtml
```

would be used to generate the email.

Variables defined in the model are accessible as instance variables in the view.

Emails by default are sent in plain text, so a sample view for our model example might look like this:

```
 Hi <%= @account.name %>,
 Thanks for joining our service! Please check back often.
```

Sending HTML Mail

To send mail as HTML, make sure your view (the .rhtml file) generates HTML and set the content type to html.

```
class MyMailer < ActionMailer::Base
    def signup_notification(recipient)
      recipients recipient.email_address_with_name
      subject    "New account information"
      body       "account" => recipient
      from       "system@example.com"
      content_type "text/html"   #Here's where the magic happens
    end
  end
```

Multipart Mail

You can explicitly specify multipart messages:

```
class ApplicationMailer < ActionMailer::Base
    def signup_notification(recipient)
      recipients      recipient.email_address_with_name
      subject         "New account information"
      from            "system@example.com"

      part :content_type => "text/html",
        :body => render_message("signup-as-html", :account => recipient)

      part "text/plain" do |p|
        p.body = render_message("signup-as-plain", :account => recipient)
        p.transfer_encoding = "base64"
      end
    end
  end
```

Multipart messages can also be used implicitly because ActionMailer will automatically detect and use multipart templates where each template is named after the name of the action, followed by the content type. Each such detected template will be added as separate part to the message.

For example, if the following templates existed:

- `signup_notification.text.plain.rhtml`
- `signup_notification.text.html.rhtml`
- `signup_notification.text.xml.rxml`
- `signup_notification.text.x-yaml.rhtml`

Each would be rendered and added as a separate part to the message, with the corresponding content type. The same body hash is passed to each template.

Attachments

Attachments can be added by using the attachment method.

Example:

```
class ApplicationMailer < ActionMailer::Base
    # attachments
    def signup_notification(recipient)
      recipients      recipient.email_address_with_name
      subject         "New account information"
      from            "system@example.com"

      attachment :content_type => "image/jpeg",
        :body => File.read("an-image.jpg")

      attachment "application/pdf" do |a|
        a.body = generate_your_pdf_here()
      end
    end
  end
```

Configuration Options

These options are specified on the class level, like ActionMailer::Base.template_root = "/my/templates"

- `template_root`: template root determines the base from which template references will be made.
- `logger`: the logger is used for generating information on the mailing run if available. Can be set to nil for no logging. Compatible with both Ruby's own Logger and Log4r loggers.
- `server_settings`: Allows detailed configuration of the server:
  - `:address` Allows you to use a remote mail server. Just change it from its default "localhost" setting.
  - `:port` On the off chance that your mail server doesn't run on port 25, you can change it.
  - `:domain` If you need to specify a HELO domain you can do it here.
  - `:user_name` If your mail server requires authentication, set the username in this setting.
  - `:password` If your mail server requires authentication, set the password in this setting.
  - `:authentication` If your mail server requires authentication you need to specify the authentication type here. This is a symbol and one of :plain, :login, :cram_md5
- `raise_delivery_errors`: whether or not errors should be raised if the email fails to be delivered.
- `delivery_method`: Defines a delivery method. Possible values are :smtp (default), :sendmail, and :test. Sendmail is assumed to be present at "/usr/sbin/sendmail".
- `perform_deliveries`: Determines whether deliver_* methods are actually carried out. By default they are, but this can be turned off to help functional testing.
- `deliveries`: Keeps an array of all the emails sent out through the Action Mailer with delivery_method :test. Most useful for unit and functional testing.
- `default_charset`: The default charset used for the body and to encode the subject. Defaults to UTF-8. You can also pick a different charset from inside a method with @charset.
- `default_content_type`: The default content type used for the main part of the message. Defaults to "text/plain". You can also pick a different content type from inside a method with @content_type.
- `default_mime_version`: The default mime version used for the message. Defaults to nil. You can also pick a different value from inside a method with @mime_version. When multipart messages are in use, @mime_version will be set to "1.0" if it is not set inside a method.
- `default_implicit_parts_order`: When a message is built implicitly (i.e. multiple parts are assembled from templates which specify the content type in their filenames) this variable controls how the parts are ordered. Defaults to ["text/html", "text/enriched", "text/plain"]. Items that appear first in the array have higher priority in the mail client and appear last in the mime encoded message. You can also pick a different order from inside a method with @implicit_parts_order.

Examples

This section is a collection of useful Rails examples.

Step By Step

How to Add a New Table

```
script/generate model <Name>
```

Generate the empty model and migration file.

```
vi db/migrate/XXX_create_<Name>.rb
```

Add columns to the table.

```
rake db:migrate
```

Migrates the data level - that is - creates the new database table.

```
vi app/models/<Name>.rb
```

Define the validations, sizes, etc.

```
vi test/unit/<Name>_test.rb
```

Define the unit tests that exercises the model validations.

If there will be a controller (and views) associated with this Model:

```
script/generate controller <Name> <action_one> <action_two> ...
```

Creates the controller and creates a view for each action.

Find

The find method of ActiveRecord is documented in the Rails API manual

`pet = Pet.find(pet_id)` Find record by **id** (an integer). **Note:** Returns one object.

`pets = Pet.find(:first, :conditions => ["owner_id = ?", owner_id])` - returns the **first** matching record. [Note: Returns one object.]

`pets = Pet.find(:all, :conditions => ["owner_id = ?", owner_id])` - find all records with a given **field value**. [Notes: 1. Returns an array of objects. Check for no records found with: `pets.empty?`. 2.`:conditions =>` supplies an SQL fragment used with *WHERE *]*

`pets = Pet.find(:all, :conditions => ["owner_id = ? AND name = ?", owner_id, name])` - find all records matching **multiple field values**. [Note: `OR` also works.]

`pets = Pet.find(:all, :conditions => ["name LIKE ?", "Fido%"])` - find all records **matching a pattern**. Wild cards are `%` for zero or more of any character, and _ for any single character. To escape a wild card use `\%` or `\_`. The reference from MySQL for LIKE will help. On the MySQL Regex website you will find examples for using REGEX.

`pets = Pet.find(:all, :order => 'name')` - find everything and **sort result** by name.

`pets = Pet.find(:all, :limit => 10, :conditions => ["owner_id = ?", owner_id])` - returns no more than the number of rows specified by **:limit**.

`pets = Pet.find(:all, :offset => 50, :limit => 10)` - uses **offset** to skip the first 50 rows.

Rake

Migrations

`$ rake db:migrate` - migrate to latest level by executing scripts in `<app>/db/migrate`. **Note:** Migration scripts are created by `script/generate model <mod-name>`

Testing

`$ rake` - run all tests.

`$ rake test:functionals` - run the functional tests, which test the controllers.

`$ rake test:units` - run the unit tests, which test the models.

`$ test/functional/<name>_controller_test.rb` - run one functional test.

Documentation

`$ rake doc:app` - generate Ruby Docs for the application. Docs are placed at `<app>/doc/app/index.html`.

Clean Up

`$ rake log:clear` - delete all logs.

`$ rake tmp:clear` - delete temporary files.

Server

`$ script/server` - start the web server for this app. By default, the server is running in development mode. By default, it will be accessible at web address: http://localhost:3000/

`$ RAILS_ENV=test script/server` - start the web server in Test Mode.

`$ script/server -e test` - start the web server in Test Mode.

`$ script/server -e production` - start the web server in Production Mode (more caching, etc.).

Fixing Errors

can't convert Fixnum to String

`some_number.to_s` - every Fixnum has method `.to_s` to convert it to a String.

Shell Commands

Certain useful shell commands that I'm always trying to remember:

`grep -r hello .` - run grep, searching for 'hello', on every file in the tree starting with the current directory.

`tar -cvzf archive.tgz <targ_directory>` - tar up directory and compress it with gzip.

Other Resources

There is a wide variety of documentation sources for Rails, including books, websites, weblogs and much more.

Plugins

- GITHub

  GITHub offers loads of plug-ins. Rails itself is also hosted there. Be sure to take a good look at them when searching for a specific plug-in or gem.

- popular Plugins

  Take at look at this list with pretty popular Rails plug-ins when searching for more common tasks

A large portion of the power of Rails comes from the wide range of plugins which are available for download.

Websites

- Rails API Documentation
- Rails Wiki

  The official wiki for Rails, not yet complete, but it gets better all the time
- Ruby on Rails plugin directory
- Learning Ruby on Rails
- Rails Documentation

  The official ROR site with everything you need to get started: guides, wikis and the very well documented API
- Videocasts for Rails

  Very popular site with advanced and many in-sight video casts. Be sure to check this site out. It offers great content and well presented videos!
- Rails for PHP

  If you have problems wrapping your mind around some of the Ruby commands nad have experience in PHP, this site will feel like heaven. Most of the most often used PHP functions are listed there with the proper Ruby/Ruby on Rails equivalent

Books

- Agile Web Development with Ruby on Rails

Development Tools

- Rails Editor

  Are you jealous of the great editor used on Railscasts? Not working an a Mac? This is the program for you: features the same functionality, look&feel and provides excellent Rails support. [Commercial - Windows]
- Open Source Development Suite Aptana

  comes with nice but complex Rails support, all Rake tasks are integrated into this IDE and you can easily switch between matching [Open Source and Commercial - All platforms]
- Editor for Rails

  Highly featured editor with rather new Rails support [Commercial - All plattforms]
- All-in-one IDE NetBeans

  offers Rails support out of the box. Support for many other languages and many plug-ins [Free - All Platforms]
- Rails and Eclipse

  If you want to work with eclipse, get RadRails. Note that the last update was submitted 2006. If you need an eclipse-like environment, try Aptana. It is built upon the eclipse platform.

Weblogs

Weblogs are one of the best ways to find out what you can do with rails - and if you know what you're looking for google can help you find them. That said, there are quite a few that are worth reading regularly.

- Ryan's Scraps

  Insights into the framework from a core team member. Covers features that are new in edge.
- Nuby on Rails

  Centers on the web design aspects of rails.

Mailing List

The Ruby on Rails Talk mailing list is a good place to ask questions: RubyOnRails-Talk Mailing List

Freenode ROR Channel

The Freenode IRC channel for Ruby on Rails is a good place to talk Rails with other developers. Please try to keep the signal-to-noise ratio at a minimum though: RubyOnRails IRC Channel

Category: Ruby on Rails