Ruby on Rails

By default, Ruby on Rails uses SQLite, but the following databases are supported:

- MySQL
- PostgreSQL
- SQLite
- SQL Server
- Sybase
- Oracle
- DB2

To specify a different database, declare it using -d [database] when creating your app and Rails will generate the appropriate type of database configuration file automatically:

```
rails –d mysql app_name
```

The database configuration file is located in app_name/config/database.yml.

If you're using any database other than the default (e.g., NOT SQLite), you may need to edit the database.yml file to specify your configuration.

Getting Started with Rails 3

This document shows using rvm to get started with Rails, but it is perfectly fine to install Ruby yourself, but be sure to also install the latest RubyGems. If you aren't using rvm, check your versions before you start.

Check Your Dependencies (not rvm)

You need Ruby 1.8.7 or later for Rails 3.

```
$ ruby -v
ruby 1.8.7 (2010-08-16 patchlevel 302) [i686-darwin10.4.0]

$ gem -v
1.3.7
```

You need RubyGems 1.3.7 or later to use bundler (which is required by Rails 3). If you have an earlier version of RubyGems you can update with:

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

Check Your Dependencies (with rvm)

With rvm you want to not only make sure you have the right version of Ruby, but also set up the directory where you will be working to always establish the right dependencies when you cd into it. You can do this with a gemset and a .rvmrc file.

```
$ rvm install 1.8.7
$ mkdir rails3_187
$ cd rails3_187
$ rvm use 1.8.7
$ rvm gemset create rails3
$ rvm use 1.8.7@rails3
$ echo "rvm use 1.9.2@rails3" >> .rvmrc
```

When you have .rvmrc project file in a directory, rvm will automatically switch to that version of ruby and the (optionally) specified gemset when you cd to that directory.

For Jruby

```
$ rvm install jruby
$ mkdir rails3_jruby
$ cd rails3_jruby
$ rvm use jruby
$ rvm gemset create rails3
$ rvm use jruby@rails3
$ echo "rvm use jruby@rails3" >> .rvmrc
```

Understanding Your Baseline

Rails is a gem. Before you start, it is helpful to know what gems you already have:

```
$ gem list
*** LOCAL GEMS ***

bundler (1.0.0)
rake (0.8.7)
```

...and where they are installed:

```
$ gem list -d

*** LOCAL GEMS ***

bundler (1.0.0)
    Authors: Carl Lerche, Yehuda Katz, André Arko
    Rubyforge: http://rubyforge.org/projects/bundler
    Homepage: http://gembundler.com
    Installed at: /Users/sarah/.rvm/gems/ruby-1.8.7-p302@global

    The best way to manage your application's dependencies

rake (0.8.7)
    Author: Jim Weirich
    Rubyforge: http://rubyforge.org/projects/rake
    Homepage: http://rake.rubyforge.org
    Installed at: /Users/sarah/.rvm/gems/ruby-1.8.7-p302@global

    Ruby based make-like utility.
```

Installing Rails

You can check which version of Rails you have with:

```
gem list rails
```

If you don't have Rails installed, install it now.

If you need your installation to go quickly, you can install without docs:

```
gem install --no-rdoc --no-ri rails
```

When working with an existing Rails 3 project, you can then install all of the dependent gems with:

```
bundle install
```

Ruby 1.8.7 note

If you want complete docs with Ruby 1.8.7 and Rails 3, you need to update rdoc first (and then update the docs of any previously installed gems) since the new version of Rails expects the new version of rdoc .

```
gem install rdoc-data rdoc
rdoc-data --install
gem rdoc --all --overwrite
gem install rails
```

You don't have to do that with Ruby 1.9.2

Rails 2 Note

If you are working on an older project and need to install a specific version of Rails, you can specify it on the command line, like this:

```
gem install -v=2.3.5 rails
```

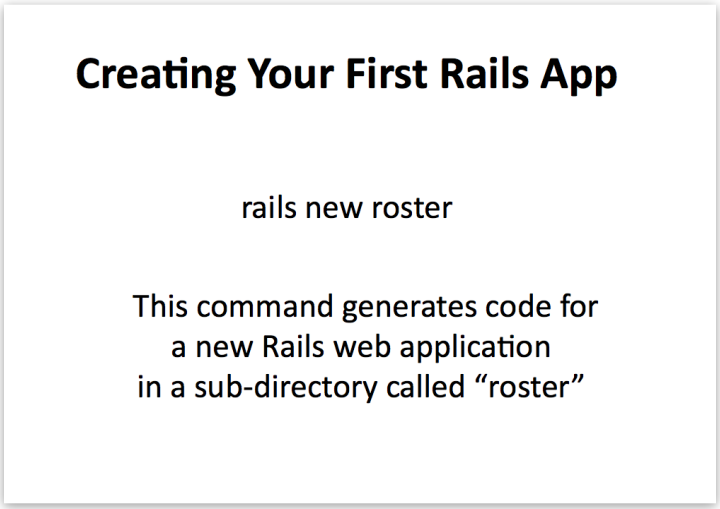
If you are using Rails 2, there is a rake task which will install all of the gems specified in the configuration file; however, that may not be all the required gems, since Rails 2 will use whatever is installed at the time it is running.

```
rake gems:install
```

When you install Rails 3 you should see something like this:

```
Successfully installed activesupport-3.0.0
Successfully installed builder-2.1.2
Successfully installed i18n-0.4.1
Successfully installed activemodel-3.0.0
Successfully installed rack-1.2.1
Successfully installed rack-test-0.5.6
Successfully installed rack-mount-0.6.13
Successfully installed tzinfo-0.3.23
Successfully installed abstract-1.0.0
Successfully installed erubis-2.6.6
Successfully installed actionpack-3.0.0
Successfully installed arel-1.0.1
Successfully installed activerecord-3.0.0
Successfully installed activeresource-3.0.0
Successfully installed mime-types-1.16
Successfully installed polyglot-0.3.1
Successfully installed treetop-1.4.8
Successfully installed mail-2.2.6.1
Successfully installed actionmailer-3.0.0
Successfully installed thor-0.14.2
Successfully installed railties-3.0.0
Successfully installed rails-3.0.0
```

Rails 3 installs many more gems than Rails 2. There aren't as many new features as there are more gems, it is just more well-factored.



For our first Rails app, we'll build a simple class list to track students in a class. We'll create a rails application called “roster” by typing “rails new roster” on the command line.

This command will create the app in a sub-directory called “roster” (which is configured to use sqlite by default)

```
$ rails new roster
```

JRuby

JRuby has additional dependencies. To create a rails app with appropriate dependendencies, the JRuby team has published a handy template:

```
$ rails new roster –m http://jruby.org
```

Rails 2

In rails 2, the rails command was only used for generating the initial application, so there was no "new" command, you simple wrote:

```
$ rails roster
```

Holy generated files, Batman! If you had Ruby, Rubygems and the Rails gem installed properly, you should have a "roster" directory with dozens of files in it. The sheer number of generated files can be overwhelming at first, but we will go through all of the Rails boilerplate as we cover each concept. This initial template (along with many helper libraries) is how Rails implements convention over configuration.

The "rails new" command creates the app in a sub-directory called “roster”

The Application Directory

- The whole rails app is in this one directory
  - No hidden configuration files in system directories
  - You will modify many of these files in the course of your development
  - We're using sqlite so even the database is in this directory, but usually the database is the only part of your application that lives somewhere else
- You can simple copy this directory to server to deploy the app
- You can delete the directory and its contents if you want to throw it away and start over

All the commands that we will run will be in the context of the application directory, so be sure to change your working directory there.

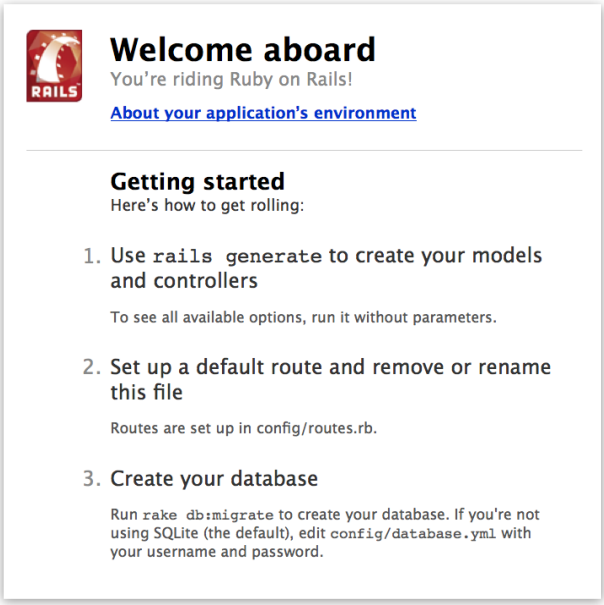
```
$ cd roster
```

Then we use the "bundle" command (short for "bundle install" from the Bundler gem) to check and install gem dependencies:

```
$ bundle
```

Now we can run our app!

```
$ rails server
```



Point your browser at http://localhost:3000 and see your running webapp! You'll see the default index of the generated rails web app containing information about your app's configuration and links to documentation.

About Bundler

Bundler is a Ruby library for managing gem dependencies. Bundler is required for Rials 3, but may be optionally used by Rails 2 (or any Ruby project).

The "bundle" command will install all dependent gems specified in the Gemfile.

Note: Rails 3 will not use any gem unless it is loaded by bundler. (Prior versions of Rails would use any gem installed on your system. While the Rails 3 restriction might seem inconvenient at first, it is very helpful when you inevitably deploy your application to a different machine.)

Save a Baseline with Git

It is valuable to keep a local repository with key points in your code, so that you can easily isolate errors and understand your progress. We'll do this using git:

```
$ git init
```

This creates a local repository for this directory (and its sub-directories). This local git repository is stored entirely within it your "roster" directory in a subfolder called ".git" which you can see with:

```
$ ls –a
```

which shows invisible files (ones that start with '.').

Next we'll want to tell git to ignore some files. Create a file called .gitignore at the root of your "roster" directory, so that you don't track the log files and temporary files in the source code repository. This is an optional, but useful, step.

Sample .gitignore

```
*.sqlite3
log/
*.kpf
*.idea
*~
.DS_Store
```

Note: You can create a a .gitignore file in your user directory (~/.gitignore) for commonly ignored files and then you don't have to create one of these files for every project.

Next add all the files (recursively) and you can use git's status command to preview what will be committed.

```
$ git add .
```

You can look at the status of what files are staged and ready to be committed:

```
$ git status
```

Lastly, commit the files to the local repository:

```
$ git commit –m "new rails app"
```

Now you can see the history of your commits annotated with your message:

```
$ git log
commit 3c5aeb908ff21df9067f806fc5d162a3f074ae9b
Author: Sarah Allen <sarah@ultrasaurus.com>
Date:   Sun Sep 26 10:59:08 2010 -0700

    new rails app
```

After files are added initially, there is a shortcut for committing all the modified or removed files that are already tracked:

```
$ git commit -am "my new stuff"
```

Modify the Index Page

You'll find the default index page in the "public" directory. All static content, such as images, javascript, css, error pages and other plain old html files live here. Edit index.html: modify title or some text, then reload http://localhost:3000 to see your changes.

Deploy Your App to the Web

We'll use Heroku for deployment since it is easy and integrates with git. Create an account now if you haven't already, at: http://heroku.com If you don't have an ssh key, you will need to make one. Heroku distributes a gem for accessing their service:

```
$ gem install heroku
```

If you are on Windows, you will need to edit your Gemfile to deploy. (If you were on a non-Windows server you could run bundle install on the server to re-install the native sqlite-ruby gem.) For deploying on Heroku, find the line where the sqlite3 gem is declared and surround it with a group statement, like this:

```
group :development, :test do
  gem 'sqlite3-ruby', :require => 'sqlite3'
end
```

Then you can deploy:

```
$ heroku create
$ git push heroku master
```

Rails Environments

Rails is configured by default with 3 environments:

- development
- test
- production

This is specified with RAILS_ENV, which can be set as an environment variable, but is "development" by default.

In your code, you refer to Rails.env (or RAILS_ENV in Rails 2)

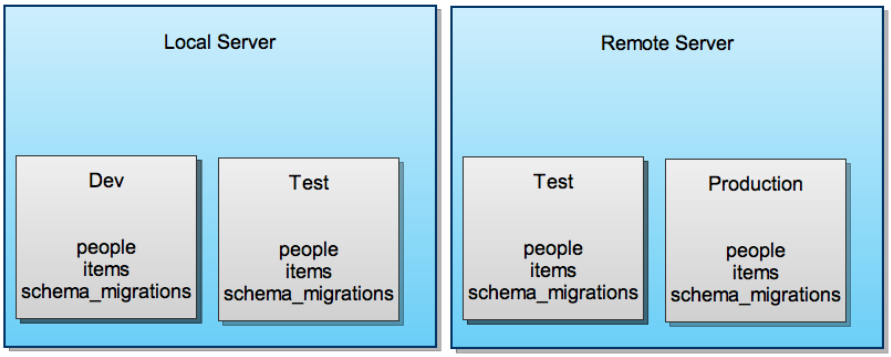
config/database.yml

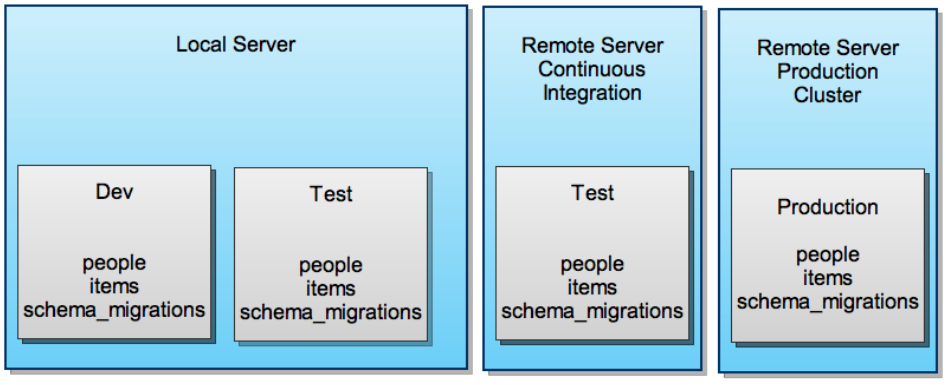
```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

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

config/environments/

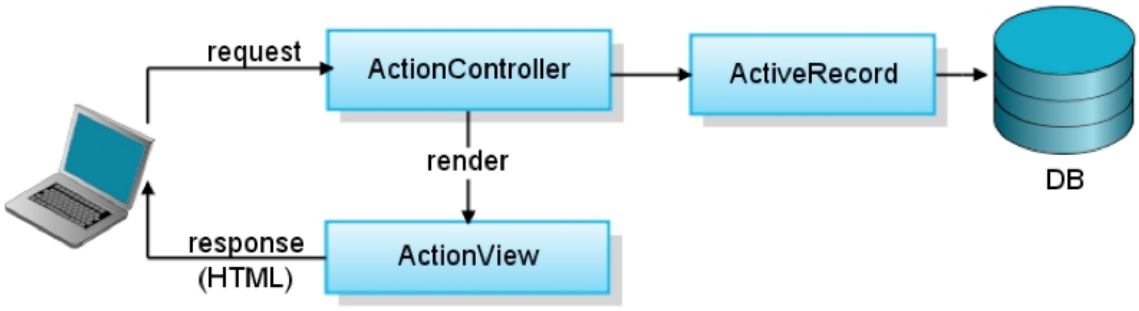
- development.rb
- production.rb
- test.rb

Common database configurations





Model View Controller



Model (ActiveRecord) represents what is in the database

View (ActionView, erb) is *typically* a model rendered as HTML

Controller (ActionController) receives HTTP actions (GET, POST, PUT, DELETE) and then decides what to do, typically rendering a view

Scaffold

Model

- app/models/person.rb
- db/migrate/20090611073227_create_people.rb

Views

- app/views/people/index.html.erb
- app/views/people/show.html.erb
- app/views/people/new.html.erb
- app/views/people/edit.html.erb
- app/views/people/_form.html.erb

Controller

- app/controllers/people_controller.rb
- route map.resources :people

Getting Started Quickly with Scaffold

Rails makes it easy to develop web pages and application code for standard database operations: create, read, update, and delete.

We'll use the rails "generate scaffold" script to generate code for creating a database table, views, for accessing the database and responding to HTTP requests.

```
$ rails generate scaffold person first_name:string last_name:string
      invoke  active_record
      create    db/migrate/20100926184235_create_people.rb
      create    app/models/person.rb
      invoke    test_unit
      create      test/unit/person_test.rb
      create      test/fixtures/people.yml
       route  resources :people
      invoke  scaffold_controller
      create    app/controllers/people_controller.rb
      invoke    erb
      create      app/views/people
      create      app/views/people/index.html.erb
      create      app/views/people/edit.html.erb
      create      app/views/people/show.html.erb
      create      app/views/people/new.html.erb
      create      app/views/people/_form.html.erb
      invoke    test_unit
      create      test/functional/people_controller_test.rb
      invoke    helper
      create      app/helpers/people_helper.rb
      invoke      test_unit
      create        test/unit/helpers/people_helper_test.rb
      invoke  stylesheets
      create    public/stylesheets/scaffold.css
```

Rails 2

In rails 2, rails scripts were not executed with the rails command, instead you simply ran the script from the script directory:

```
$ script/generate scaffold person first_name:string last_name:string
```

on Windows:

```
$ ruby script/generate scaffold person first_name:string last_name:string
```

the ruby command is required for executing ruby scripts.

If you use any other database (except for sqlite) you need to create the database:

```
$ rake db:create:all
```

To apply the code and set up the database:

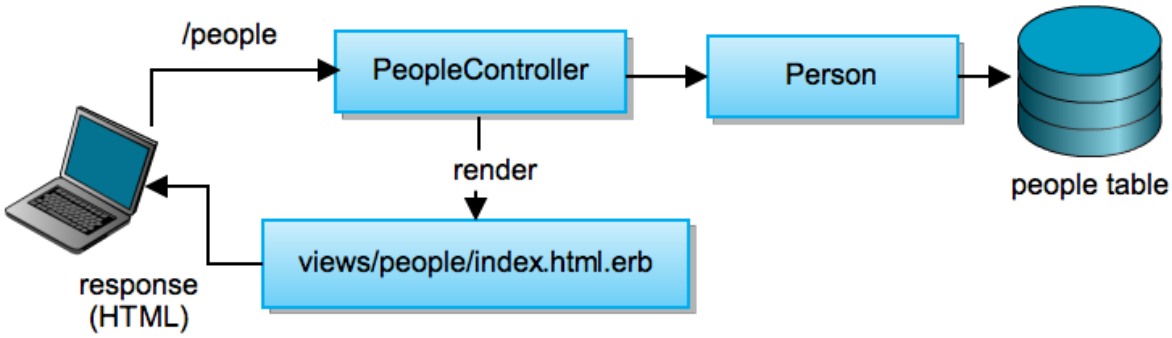
```
$ rake db:migrate
```

Run the server:

```
$ rails server
```

With a browser, go to http://localhost:3000/people

Naming Conventions



Default names in Rails take advantage of helper methods to pluralize the model name to create the database table name and handle camel case for Class names and underscores to separate words in filenames.

- Models are singular (e.g. Person, Book)
- Database table that stores a set of models is plural (e.g. people, books)
- Controllers are plural (e.g. PeopleController, BooksController)

A nice interactive demonstration of Rails pluralization: http://nubyonrails.com/tools/pluralize

Model

The model represents a single object (which maps to a single row in the corresponding database table). You'll find the code in app/models/person.rb

```
class Person < ActiveRecord::Base
end
```

Database

In your database, a table stored the data for a collection of objects. You keep data in a "people" table:

```
rails dbconsole
>> select * from people;
```

Controller

The controller works with various models so it is plural. You'll find the code in app/controllers/people_controller.rb

```
class PeopleController < ApplicationController
end
```

Pluralization

Pluralization rules are controlled by the ActiveSupport::Inflector class in Rails. Since 2.x, Rails has provided a default initializer with comments that indicate exactly how to configure this (before that you would have had to just look it up and then configure in config/environment.rb).

Note that the irregular noun "foot" is not correctly pluralized in Rails.

```
> "foot".pluralize
=> "foots"
```

To fix this, edit the file config/initializers/inflections.rb changing the commented out person/people line to foot/feet:

```
ActiveSupport::Inflector.inflections do |inflect|
#   inflect.plural /^(ox)$/i, '\1en'
#   inflect.singular /^(ox)en/i, '\1'
    inflect.irregular 'foot', 'feet'
#   inflect.uncountable %w( fish sheep )
end
```

Now when you go back to the console (after exiting and starting again, "reload!" is not sufficient in this case) you'll see the correct pluralization:

```
$ rails console
Loading development environment (Rails 3.0.0)
 >> "foot".pluralize
 => "feet"
```

Adding Data Interactively

With a browser, go to http://localhost:3000/people

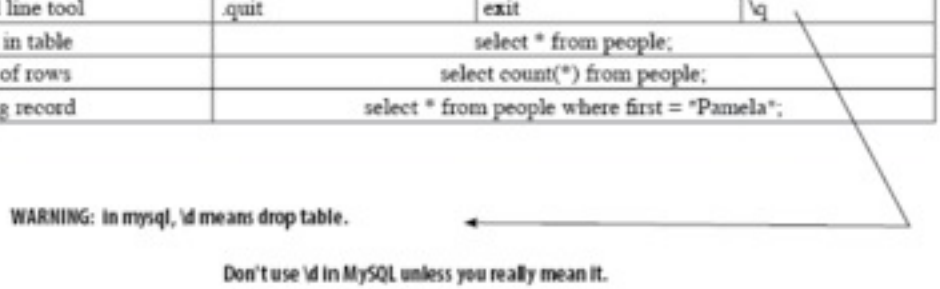
Experiment: add 6-10 people to the list,
delete some people
edit some people

See What is Happening in the Database

```
$ rails dbconsole

>> select * from people;
```

Database Command Reference

	sqlite	MySql	PostgreSQL
list tables in current db	.tables	show tables;	\dt
show SQL for table create		show create table people;	n/a
list columns	.schema people	describe people;	\d
exit command line tool	.quit	exit	\q
show all rows in table	select * from people;		
show number of rows	select count(*) from people;		
show matching record	select * from people where first = "Pamela";		



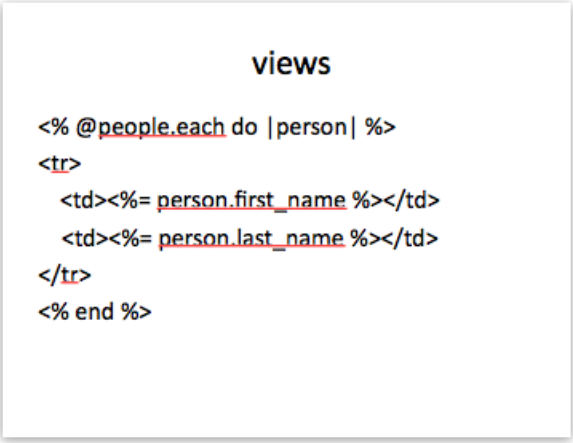
Understanding how Rails works with the Database

Let's talk a closer look at how Rails works with the database. We'll start from the beginning and examine the results of each step. We'll start with a fresh rails app, so you can see what happens in slow motion.

1) rails new test_app → no database, not much in db directory
2) rake db:create:all → database is created (but not for Sqlite), no tables
3) rake db:migrate → a schema migrations table with nothing in it
   db/schema.rb
4) rails generate model ... → just source code
   db/migrate/...
5) rake db:migrate → a row in schema migrations and a table
   db/schema.rb

Editing views

In app/views/people/index.html, you will see code that is HTML mixed with Ruby that looks something like this:



Note: the above is a subset of the index file to simplify the example.

ERb = Embedded Ruby files

<% ... %> executes ruby code (e.g., setting up conditionals)

<%= ... %> evaluates ruby code, inserts result in HTML (e.g., displaying value returned from a method)

Rails 2			
<%= h(@person.name) %>			HTML escape text
<%-	...	%>	executes ruby code, omits blank line in HTML output

Exercise

1. On the main people page
    a. Change “Listing people” to “My Class List”
    b. List people with first initial and last name in one visual column (e.g. J. McDonald)

ActiveRecord

Next we'll explore ActiveRecord, which is the “model” in the Rails Model-View-Controller pattern. Rails uses ActiveRecord to represent data in a relational (SQL) database. For those who know the term, it is an ORM (Object Relational Mapper). Note that you can write a Rails app that doesn't use a database at all, or you can work with other kinds of data stores, such as "no sql", document-oriented databases. However, Rails was originally created for developing a frontend to a relational database, and that’s still the most common use case for web applications.

ActiveRecord allows you to:

- work with Ruby Objects, instead of database tables and records
- write very concise code to fetch and store data
- easily write and maintain database independent code

You’ll typically have an ActiveRecord model for every database table. For example, in the last chapter, we created a "person" object with the "generate scaffold" command. This generated a Person class, which we call a model, and a corresponding "people" table. The model represents a single row in the database table.

Now we’ll explore Active Record::Base methods using the Person model. It is critical to understand the behaviors that are inherited from ActiveRecord before writing our own code. We will use interactive ruby in the rails console to learn about how ActiveRecord works. This form of interactive testing will help you establish a good understanding of the base class. This makes it so when you write your own code you can test-drive the creation of your code's unique behavior and not worry about the built-in functionality (which is already tested in the Rails core). It is important in the design phase (when writing your tests) to focus on just your added behavior, which will lead you to be able to leverage Rails in powerful ways.

We will then learn how to customize the model, test-driving those changes, including how to modify the model's data structure using migrations.

Introducing Rails Console

The rails console runs irb, but it also loads your current application environment. You will have access to the same classes that are available in your rails app.

[Note: this is completely different from dbconsole, which accesses your database console.]

Launch the console by executing the following command inside the "roster" application main directory:

```
rails console
```

As a shortcut you can also type

```
rails c
```

To better understand how ActiveRecord works, we’ll look at SQL queries logged by Rails when the database is accessed by ActiveRecord.

Typically you will look at the log with:

```
tail –f log/development.log
```

When learning, it is useful to see the log output interleaved interactively with your Ruby code. To do this in rails console, type:

```
>> ActiveRecord::Base.logger = Logger.new(STDOUT)
```

This will echo all of the log output to standard out in addition to the log file.

Experimenting with ActiveRecord

We’ll start by creating an instance of the Person class. Remember the Person model is simply a Ruby class, so we can create an instance in the console just like we would create any other Ruby object:

```
>> bret = Person.new
```

All of the columns in the people database table are available as attributes of the instance:

```
>> bret.first_name = “Bret”
>> bret.last_name = “Smith”
>> bret
=> #<Person id: nil, first_name: “Bret”, last_name: “Smith”, present: nil,
created_at: nil, updated_at: nil>
```

If you try to access an attribute that is not defined in the people table, a NoMethodError will be raised:

```
>> bret.xxx
NoMethodError: undefined method `xxx' for #<Person:0x0000010290a5d0>
      from /Users/sarah/.rvm/gems/ruby-1.9.2-head@rails3/gems/
activemodel-3.0.0/lib/active_model/attribute_methods.rb:364:in
`method_missing'
```

We can also supply attributes when calling the constructor:

```
>> may = Person.new(:first_name => "May", :last_name => "Fong")
 => #<Person id: nil, first_name: "May", last_name: "Fong", present: nil,
created_at: nil, updated_at: nil>
```

Note that we have just created two objects that exist only in-memory – nothing has yet been saved to the database. You can see that the id is nil, indicating an unsaved record. You can also call:

```
>> may.new_record?
=> true
```

To save to the database:

```
>> may.save
  SQL (0.3ms)   SELECT name
 FROM sqlite_master
 WHERE type = 'table' AND NOT name = 'sqlite_sequence'

  SQL (11.9ms)  INSERT INTO "people" ("created_at", "first_name",
"last_name", "present", "updated_at") VALUES ('2010-09-06 07:09:18.238744',
'May', 'Fong', NULL, '2010-09-06 07:09:18.238744')
 => true
>> bret.save
```

The save method returns true if the record is successfully saved and false if there is an error. Now you can see that new_record? will return false and that *may* now has values for id, created_at, and updated_at, which were automatically assigned when the record was saved. Also note that the "bret" record can also be saved. Whether you create an instance of the Person class with attributes passed to the constructor or set them later, the instance will save the current value of that attribute.

```
>> may.new_record?
=> false
>> may
 => #<Person id: 1, first_name: "May", last_name: "Fong", present: nil,
created_at: "2010-09-06 07:09:18", updated_at: "2010-09-06 07:09:18">
```

ActiveRecord tracks whether any attributes have been modified, so that if you call save again, it won't actually call the database, since it knows that the record has note changed.

```
>> may.save
 => true
```

You can look at individual attributes to see what has changed:

```
>> may.last_name
 => "Fong"
>> may.last_name_changed?
 => false
>> may.last_name = "Woo"
 => "Woo"
>> may.last_name_changed?
 => true
>> may.changed?
 => true
>> may.save
  SQL (40.4ms)  UPDATE "people" SET "last_name" = 'Woo', "updated_at" =
'2010-09-06 07:14:08.497625' WHERE ("people"."id" = 1)
 => true
```

Then when you have modified attributes, if you save the record, it will call the database with an "UPDATE" instead of the "INSERT" that was triggered by saving the new record.

In Rails, there is a shortcut that does both new and save in a single operation. The create method is frequently used to provide a list of attributes and save them to the database:

```
>>john = Person.create(:first_name => "John", :last_name => "Woo")
  SQL (0.4ms)  INSERT INTO "people" ("created_at", "first_name", "last_name",
"present", "updated_at") VALUES ('2010-09-06 20:46:13.660208', 'John', 'Woo',
NULL, '2010-09-06 20:46:13.660208')
 => #<Person id: 3, first_name: "John", last_name: "Woo", present: nil,
created_at: "2010-09-06 20:46:13", updated_at: "2010-09-06 20:46:13">
```

In another window, run rails dbconsole and rails server. Notice that when you call save in rails console, you are working with the live environment, and your changes can be seen in your database and in your web application:

```
$ rails dbconsole
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from people;
1|May|Woo||2010-09-06 07:09:18.238744|2010-09-06 07:14:08.497625
2|Bret|Smith||2010-09-06 07:46:53.214065|2010-09-06 07:46:53.214065
3|John|Woo||2010-09-06 20:46:13.660208|2010-09-06 20:46:13.660208
```

You should have these three records in your database to follow the examples in the next section.

ActiveRecord Queries

ActiveRecord allows you to find by id or find first, last or all records. Any of these methods will trigger a database SELECT every time they are called.

```
Person.find(1)    # SELECT * FROM people WHERE id=1;
Person.all        # SELECT * FROM people
Person.first            # SELECT * FROM people limit 1
Person.last             # SELECT * FROM ORDER BY id DESC limit 1
```

TIP: To display in the console in yaml format (sometimes easier to read):

```
y Person.all
```

Conditions

You can specify literal SQL as a condition; however, using a hash is easier and safer since Rails will worry about quoting, escaping and protecting you from SQL injection attacks.

```
Person.where(:first_name => 'May')
# SELECT * FROM people WHERE first_name='May';

Person.where(:first_name => 'May', :last_name => 'Woo')
# SELECT * FROM people WHERE first_name='May' AND last_name='Woo';
```

Dynamic Queries

ActiveRecord (as of Rails 3) has a dynamic query language. The "where" method along with other modifiers (such as order, limit, etc.) returns an ActiveRecord::Relation object. The SQL query is not actually performed until the result is required (typically when an enumerable method, such as "each" is called on the result).

Examples:
Person.where(:last_name => "Woo").order(:first_name)
Person.where(:last_name => "Woo").order(:first_name).limit(1)

# order may be called with a string or a symbol
Person.order("first_name")

# you can optionally specify ASC or DESC when calling order
Person.order("last_name DESC").limit(2)
Person.limit(2)

# you can call these methods in any sequence
Person.order("first_name").where(:last_name => "Woo")
Person.where(:last_name => "Woo").order("first_name")

For the curious...

Why Do I See Queries in irb?

When I use irb, I see the results right away, why?

```
> result = Person.where(:first_name => "May")
  Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
("people"."first_name" = 'May')
 => [#<Person id: 1, first_name: "May", last_name: "Woo", present: nil,
created_at: "2010-09-06 07:09:18", updated_at: "2010-09-06 07:14:08">]
```

Irb always calls inspect on whatever is returned and then prints that to the console. Inspect will trigger ActiveRelation to execute its SQL query so that it can report the results. To see this, try:

```
> result = Person.where(:first_name => "May"); nil
 => nil
> result.class
 => ActiveRecord::Relation
> result
  Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
("people"."first_name" = 'May')
 => [#<Person id: 1, first_name: "May", last_name: "Woo", present: nil,
created_at: "2010-09-06 07:09:18", updated_at: "2010-09-06 07:14:08">]
```

For more on ActiveRecord queries see: http://edgeguides.rubyonrails.org/active_record_querying.html#retrieving-objects-from-the-database

Using ActiveRecord in Your Web App

Now that you understand a bit about ActiveRecord, let's look at our application again in the browser. Start the web app:

```
rails server
```

When you browse to http://localhost:3000/people you should see:

Note that all of the changes that you made in the database appear in the web application, whether you made changes in rails console via ActiveRecord or if you made changes using SQL in dbconsole. These are just three different ways of looking at and interacting with the same database.

Notice that you can click New Person, not fill in anything, click "Create Person" and this default Rails application will happily create a completely blank record; however, you will typically want to set up some checks to make sure people add good data. Since this is such a common part of web applications, Rails makes this easy.

For example, we may want to make sure that a last name is provided. To do this, we can make a simple change to the model. Open app/models/person.rb and notice that the implementation is a simple subclass of ActiveRecord::Base. All of the behavior that we've seen so far is built into ActiveRecord. As we've seen the attributes are dynamically discovered from the structure of the

corresponding database table. If we want to require specific attributes to be present or in a specific format before saving, we add a validation to the model.

```
class Person < ActiveRecord::Base
        validates_presence_of :first_name
    end
```

With this change, if we attempt to save a blank record, it fails to save and the error will be reported:

Rails provides a lot of behavior in just one line of code. To understand exactly what is going on, let's go back to the console. (Note: unlike when you are running Rails as a server, in the console your code doesn't automatically reload. You need to restart rails console or use the reload! command to reload your rails app.)

```
>> person = Person.new
 => #<Person id: nil, first_name: nil, last_name: nil, present: nil,
created_at: nil, updated_at: nil>
>> person.save
    => false
```

Save now returns false. We can inspect what failed by looking at the errors on the person object.

```
>> person.errors
=>{:first_name=>["can't be blank"]}
```

Note that we can also test for validation errors without attempting to save. Calling valid? will also set errors if one or more validations fail.

```
>> person.valid?
=> false
```

If a validation fails, save or create will return false. To save the person object and throw an exception if there is an error, you can call the alternate methods: save! or create!

```
> person.save!
ActiveRecord::RecordInvalid: Validation failed: First name can't be blank
...stack trace follows...
```

Declarative Validations

validates_presence_of

```
validates_presence_of :first_name
validates_presence_of :password_confirmation, :if => :password_changed?
```

validates_length_of / validates_size_of

```
validates_length_of :first_name, :maximum => 30
validates_length_of :last_name, :maximum => 30,
                    :message=>"{{count}} is too long"
validates_length_of :fax, :within => 7..32, :allow_nil => true
validates_length_of :phone, :within => 7..32, :allow_blank => true
validates_length_of :user_name, :within => 6..20,
                    :too_long => "pick a shorter name",
                   :too_short => "pick a longer name"
```

validates_inclusion_of

```
validates_inclusion_of :gender, :in => %w( m f )
validates_inclusion_of :age, :in => 0..99
validates_inclusion_of :format, :in => %w( jpg gif png ),
      :message => "extension {{value}} is not included in the list"
```

validates_format_of

```
validates_format_of :email, :with => /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-
z]{2,})\Z/i,
```

validations for common UI patterns

terms of service:	`validates_acceptance_of`
email/password validation:	`validates_confirmation_of`

and more!

```
validates_numericality_of,
validates_uniqueness_of,
validates_exclusion_of,
validates_associated,
validates_each
```

Test Driving Validations

Now that you understand the basics of ActiveRecord and how to add a validation, let's dive into the more standard test-driven approach to creating model behavior. We're taking this approach for two reasons: first, it is a good way to write code and if you learn to write code using test-driven techniques, then you will be able to use those techniques in your practice. Second, it is an effective way to learn. Rails includes a lot of "magic" which makes you powerful after you understand it, but can be very hard to learn. The Rails magic often takes place between the core objects. Much of your application code is implicit. Testing causes you to have to set up the preconditions and verify the postconditions, making the implicit behaviors of Rails explicit. When you can do this effectively, then you really understand what is going on and that understanding is as valuable in making you a effective developer, as using testing in your practice.

To take the test-first approach, delete (or at least comment out) the validation we added to the Person model. We'll start over and write the same code after we write a test and watch it fail.

We'll use rspec, as we did for Ruby in previous chapters. For rails, we also need to have the rspec-rails gem installed[1]. Add rspec-rails to the :test and :development groups in the Gemfile:

```
group :development, :test do
  gem "rspec-rails"
end
```

```
Note: It needs to be in the :development group to expose generators and rake
tasks without having to type RAILS_ENV=test.
Next on the command line, type:
rails generate rspec:install
Which will report that it has created a number of files an directories:
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  autotest
         create autotest/discover.rb
```

Next create a "models" directory under the "spec" directory. By convention, we place our tests in the same file hierarchy as our application files. This saves effort since the framework will automatically set up the correct load paths and make it so we don't have to explicitly require the model source file.

```
$ mkdir spec/models
```

The spec file must end with _spec.rb and by convention begins with the lower-case name of the class that is being tested with words separated by underscores. So to test the Person class, we create the file person_spec.rb

Create the file "spec/models/person_spec.rb" with the following contents:

```
require "spec_helper"

describe Person do

  it 'should save a valid record with first_name and last name' do
    p = Person.new(:first_name => "Eve", :last_name => "Smith")
    p.save
    p.should be_valid
  end
```

1

```
  it 'should not save a record with only first name' do
    p = Person.new(:first_name => "Eve")
    p.save
    p.should_not be_valid
    p.errors[:last_name].should include("can't be blank")
  end

  it 'should not save a record with only last name' do
    p = Person.new(:last_name => "Smith")
    p.save
    p.should_not be_valid
    p.errors[:first_name].should include("can't be blank")
  end

end
```

Rails 2

In Rails 2, each attribute has a single error, so in the above spec the last two examples would instead check:
`p.errors[:last_name].should == "can't be blank"`
and
`p.errors[:first_name].should == "can't be blank"`

RSpec Syntax note:
Any method that ends in a ? (known as a predicate) can be used automatically with rspec in a way that sounds a bit like English. Above we see that valid? can be test with person.should be_valid. RSpec provide this for all predicated. If the matcher begins with "be_", RSpec removes the "be_", appends a "?", and sends the resulting message to the given object. In fact, Rspec will do the same for be_a_ or be_an_ to support some predicates which make more sense read that way such as instance_of? or kind_of?

```
$rake spec
.FF

Failures:
  1) Person should not save a record with only first name
     Failure/Error: p.should_not be_valid
     expected valid? to return false, got true
     # ./spec/models/person_spec.rb:14:in `block (2 levels) in <main>'

  2) Person should not save a record with only last name
     Failure/Error: p.should_not be_valid
     expected valid? to return false, got true
     # ./spec/models/person_spec.rb:21:in `block (2 levels) in <main>'

Finished in 0.10642 seconds
3 examples, 2 failures
```

Now you can add validations to the Person model class to make the spec pass.

Security Advisory: Protect Your Code from SQL Injection

Randall Munroe http://xkcd.com/327/

Imagine a scenario where you have designed a login form which ends up calling a method on your model to authenticate. So, you want a User model with an authenticate that behaves like this:

```
it "should authenticate successfully with correct name and password"
    User.create!(:name => "pat", :password => "password")
    User.authenticate("pat", "password").should_not be_nil
end

it "should fail to authenticate with incorrect name"
it "should fail to authenticate with incorrect password"
```

Conditions can either be specified as a string, array, or hash representing the WHERE part of an SQL statement. The array form is to be used when the condition input is tainted and requires sanitization. The string form can be used for statements that don't involve tainted data. The hash form works much like the array form, except only equality and range is possible.

Let's write a test that would fail if your code was vulnerable.

```
it "should authenticate successfully with correct name and password"
    User.create!(:name => "pat'; DROP TABLE Users; --",
                 :password => "password")
    User.authenticate("pat", "password")
    User.count.should_not == 0
end
```

The following UNSAFE code would cause the test to fail:

```
      where("user_name = '#{user_name}' AND password = '#{password}'")
```

The simplest and safest way to implement conditional queries is by specifying the conditions as a hash

```
  class User < ActiveRecord::Base
    def self.authenticate(user_name, password)
      where(:user_name => user_name, :password => password )
    end
   end
```

The following code has identical behavior, but is included here to illustrate the array form which allows you to write arbitrary SQL. The first element of the array contains SQL code with a "?" wherever your want to insert text from a variable. Rails will take care of escaping the variable to make it safe.

```
      where([ "user_name = ? AND password = ?", user_name, password ])
```

Custom Model Behavior

We saw in the security advisory above that we created a class method "authenticate" to look up the user in the database and return the record if authentication was successful or nil if not. It is standard practice to extend your model class by adding class or instance methods with custom behavior. Remember that Rails is Ruby. All of the capabilities of the Ruby language are available to you within your Rails application. Your model is just a Ruby class which may be extended in standard ways to have its own methods. It is considered a best practice to add logic to the model when possible, so that you can keep your views and controllers small.

Let's look at a simple example with our person object. Suppose you commonly refer to people with their full name (concatenating first and last name).

```
it 'should have a full name' do
  p = Person.new(:first_name => "Eve", :last_name => "Smith")
  p.full_name.should == "Eve Smith"
end
```

Add this example to your person_spec.rb file and run "rake spec" and you will see the following error.

```
1)
NoMethodError in 'Person should construct a full name'
undefined method `full_name' for #<Person:0x1036ff5f0>
.../lib/active_record/attribute_methods.rb:264:in `method_missing'
./spec/models/person_spec.rb:12:
```

You can see that Ruby reports a NoMethodError, since we are sending a full_name message to person object "p" where there is no method definition (or corresponding table column for ActiveRecord to dynamically evaluate). To make this example pass, we can simply define a method:

```
class Person < ActiveRecord::Base
  def full_name
    first_name + last_name
  end
 end
```

Summary

In this section, we learned about how to use ActiveRecord to create, access and modify the data from your database. We also learned about how it supports the common web pattern of form validation. Lastly, we saw how to extend your model by writing code in the same way that we extend any class in Ruby.

Developing Your Own Migrations

To generate the template for a migration run the following script (replacing "add_something" with a descriptive name)

```
$ rails generate migration add_something
```

The rails generate command will create a file in db/migrate, where you can edit the migration code like that shown below:

```
class AddSomething < ActiveRecord::Migration

  def self.up
      add_column :people, :something, :string
  end

  def self.down
      remove_column :people, :something
  end
end
```

rake db: migrate:up migrates the database to the new state

rake db: migrate:down reverts the database to previous state.

If you find an error and want to revert the generate command, running the same command with "destroy" instead of "generate" will delete the files created in the previous step:

```
$ rails destroy migration add_something
```

To perform the migration:

```
$ rake db:migrate
```

it's a good idea to run "redo: which tests the integrity of the migration by migrating down then back up again:

```
$ rake db:migrate:redo
```

Migration Reference Sheet

When you add features to your application that require database schema changes, you will create migrations that modify the schema to support the new behavior.

Available Methods

```
create_table(name, options)
drop_table(name)
rename_table(old_name, new_name)
add_column(table_name, column_name, type, options)
rename_column(table_name, column_name, new_column_name)
change_column(table_name, column_name, type, options)
remove_column(table_name, column_name)
add_index(table_name, column_name, index_type)
remove_index(table_name, column_name)
change_table(table_name) {|Table.new(table_name, self)| ...}
```

Available Column Types

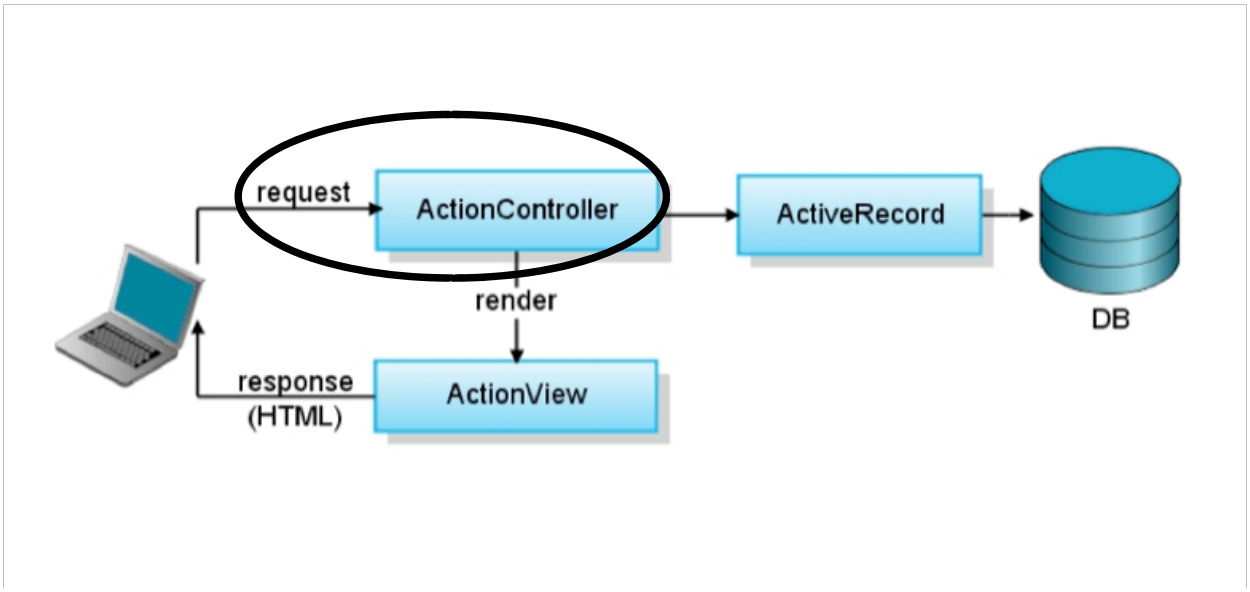
```
integer
text
float
string
datetime
binary
date
boolean
timestamp
decimal :precision, :scale
time
```

Valid Column Options

```
limit
null (i.e. ":null => false" implies NOT NULL)
default (to specify default values)
for decimal, :precision => 8, :scale => 3
```

For more information see: http://api.rubyonrails.org/classes/ActiveRecord/Migration.html

Routes and Controllers



Routes are the part of Rails that specify how URLs that come from the outside world call your code. Typically a route will match a URL to a controller action (which is simply a method of a controller class.

Controllers direct traffic in your Rails application. They take requests from the outside world (through routes) and decide what model to access or update and then what view to render, alternately they can redirect to another URL or controller action. When written correctly, they don't do anything more than that. To keep your code maintainable you want to put as little code in the controller as possible. Each method should be small, so that you can clearly look at the whole controller to see the flow of control for your application.

Routes

The mapping between URLs and code, implemented with Rails routes, serves two purposes:

- Recognize URLs and trigger a controller action
- Generate URLs from names or objects, so you don't have to hard code them in views

You specify routes in config/routes.rb. Rails 2 defines some default routes, which are relegated to comments in Rails 3.

Note: **Position in routes.rb matters**
When determining how to route something Rails will use the first applicable route it finds going through routes.rb from top to bottom. Because of this its a good practice to put more specific routes near the top and more general routes at the bottom.

Experimenting with Routes Controllers

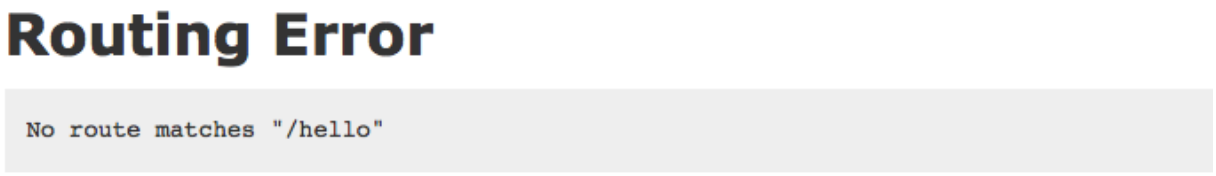
We'll start out by creating a very simple application that just responds with nice greetings like "hello" and "good morning." In order to understand what Rails gives you by default and what code is required (or optional), we'll approach this by interactively testing a new rails application.

On the command like type:

```
$ rails new message_app
$ cd message_app
$ rails server
```

Then in a web browser, go to the URL: http://localhost:3000/hello

You should see:



This makes sense because we haven't built anything in the application to support messages. Rails is telling us that the URL /hello is not defined. There is no "route" into the app based on that name. So, let's create a route!

We'll be creating a controller where our code will live that implements this, since we want to handle lots of different kinds of messages, just so you can see what's what we'll give everything a unique name. We'll plan to create a messages controller with a greeting action.

In config/routes.rb, you will see a block of code with a lot of comments in it, delete the comments and add your first route to make the file look like this:

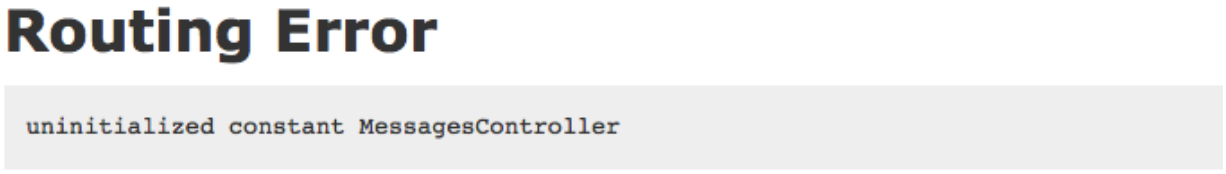
```
MessageApp::Application.routes.draw do
      match "hello" => "messages#greeting"
end
```

Note: an alternate longer form of this same route specification is:

```
      match "hello", to: => "messages#greeting"
```

These do the same thing. The first is simpler and requires less typing.

Now, we still haven't written any code yet, but let's explore what we have done by pointing our browser to http://localhost:3000/hello



Here you can see that Rails will take the first part of the route destination (before the #): messages, capitalize it, concatenate it with "Controller" and look for a class of that name. In this case, MessagesController.

Next we'll use a rails generator to create our controller class. In your terminal, type:

```
rails generate controller messages
```

And you should see the following output:

```
create  app/controllers/messages_controller.rb
invoke  erb
create    app/views/messages
invoke  test_unit
create    test/functional/messages_controller_test.rb
invoke  helper
create    app/helpers/messages_helper.rb
invoke    test_unit
create      test/unit/helpers/messages_helper_test.rb
```

The generator creates a controller and a "helper" along with a directory for the views that typically go with them and corresponding tests.

Before we dive into writing code in our controller, let's look at the default behavior of Rails now that we have addressed our previous error. Start your server again (rails server from the command line) and use your browser to go to: http://localhost:3000/hello

You can now see that Rails interprets the second part of the route destination string (after the #) as a controller action.

What's a controller action?

A controller action may take several forms:

- a view template, implemented with EmbeddedRuBy (ERB)
- a controller method that calls "render"
- a controller method that sets up instance variables for a view template, then renders the template (with an explicit or implicit render call)

Let's start with the simplest controller action by creating a view template with no additional code in your controller class.

View Template

To create an action without any code, we rely on the built-in Rails conventions. Rails expects view templates for the MessagesController to be found in the /app/views/messages directory. By default, Rails will look for a template with the same name as the action. For the route "messages#greeting" we'll create a file /app/views/messages/greeting.html.erb with some basic html, such as:

```
<h1>Here I am!</h1>
```

Now when we point our browser to: http://localhost:3000/hello we see our "Here I am!" message. You can see in the log (output in your terminal window where you are running rails server) that the browser issued a GET request for the "/hello" URL which was processed by the MessagesController greeting action to render "messages/greeting.html.erb" which took 1.8 milliseconds to render.

```
Started GET "/hello" for 127.0.0.1 at 2010-10-10 12:14:59 -0600
  Processing by MessagesController#greeting as HTML
Rendered messages/greeting.html.erb within layouts/application (1.8ms)
Completed 200 OK in 20ms (Views: 20.0ms | ActiveRecord: 0.0ms)
```

Typically you would only take this approach for the very simplest of views (or for quick mockups when prototyping). More commonly you will need to implement some business logic and prepare

some dynamic data to be rendered in the view. While you can embed any code in your template, it is a better idea to keep your views lean and focused on implementing your visual design.

Controller Logic

Look at app/controllers/messages_controller.rb and you will see that we now have a controller, which is simply a subclass of ApplicationController. Now, let's add a controller method corresponding to the "greeting" action defined in our route:

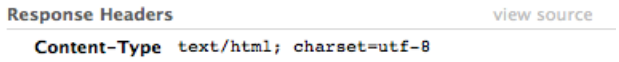
```
class MessagesController < ApplicationController
      def greeting
      end
end
```

If you go to http://localhost:3000/hello in your browser, you will see that no behavior has changed. You don't need to call "super" in your controller method. Even with no implementation, it will render the template of the same name by default.

Now let's render something different

```
class MessagesController < ApplicationController
      def greeting
            render :text => "plain old text here"
      end
end
```

Now, when you go to http://localhost:3000/hello you can see that our controller method overrode the default controller behavior.



In addition to html and text, Rails also has built in support for xml and json:

```
class MessagesController < ApplicationController
      def greeting
            render :xml => {:greeting => "hello"}
      end
end
```

```
class MessagesController < ApplicationController
      def greeting
            render :json => {:greeting => "hello"}
      end
end
```

When you want to set up dynamic data for your views you will typically use instance variables.

Controller Testing

Controller tests rely heavily on helper methods that allow you to trigger controller actions, set up state before the action and inspect the state after the action.

```
describe PersonController do
      describe "GET show" do
            it "renders" do
                  get :show, {:name => "Zak"}
                  # call the PersonController show method
                  # with params[:name] == "Zak"

                  assigns[:name].should == "Zak"
                  # tests to see if you set @name="Zak" in the show method

            response.should render_template("show")
                  # default behavior, will only fail if you called
                  #           render 'other'
      end
      end
end
```

```
class PersonController < ActionController:Base
  def show
    @name = "zak"
  end
end
```

get :show | assigns[:name]

show.html.erb

In the view template, the instance variables that are set in your controller are also made available:

```
<%= @name %>
```

get, post, put, delete

All of the HTTP actions are available as methods that you can call in your controller tests to run the action in a similar way to which is is called when accessed from a web browser (or any HTTP client).

action: name of the controller action to process

parameters: The HTTP parameters that you want to pass. This may be nil, a Hash, or a String that is appropriately encoded (application/x-www-form-urlencoded or multipart/form-data).

```
get :index
get :index, {:name => "Zak", :color => "blue"}
```

assigns
In a controller, you will typically set instance variables that are later available in the view that is rendered by that controller. In order to make the controller instance variables available as view instance variables, Rails stores these in an "assigns" hash. The assigns hash is protected and not typically accessed in your Rails code; however, for testing, it is important to know that your controller has set up instance variables correctly.

```
assigns[:name].should == "Zak"
```

should render_template
lib/spec/rails/matchers/render_template.rb
Passes if the specified template will be rendered by the response. However, in a controller spec the rendering is not actually taking place. The path can include the controller path or just the file name. The extension is optional -- it assumes .rhtml or the equivalent .html.erb.

```
response.should render_template(:list)
response.should render_template('list')
response.should render_template('list.xml.erb')
response.should render_template('same_controller/list')
response.should render_template('other_controller/list')
response.should render_template('other_controller/list.html.erb')
response.should render_template('other_controller/list.xml.erb')
```

Integration Testing – Rspec 1/Rails 2 only (always on in RSpec 2/Rails 3)
Controller specs can be run in two modes: isolation or integration. Isolation testing is described above. In isolation mode, view templates are not verified -- they are not rendered, nor does it matter whether they exist.

It is common practice to use controller specs as integration tests by specifying that the views be rendered. To do so, there is a one line directive to add to your spec: `integrate_views`. In the example below, the spec will fail if there is an error in the view, if the template does not exists, as well as if the controller renders an alternate template.

```
describe BooksController do
      integrate_views

      it "renders index" do
            get :index
            response.should render_template(:index)
      end
end
```

Understanding Controllers with Test First Development

```
$ rails new echo

$ git init
$ git add .
$ git commit -m "new app"
```

edit Gemfile, at the bottom, uncomment the development/test group:

```
group :development, :test do
  gem 'rspec-rails'
end

$ bundle
$ rails generate rspec:install
$ rails generate rspec:controller echo
```

add the following code:

```
      require 'spec_helper'

      describe EchoController do
        describe "GET index" do
          it "renders" do
            get :index
            response.should render_template('echo/index')
          end

          it "renders with text param" do
            text = "something"
            get :index, {:text => text}
            assigns[:text].should == text
            response.should render_template('echo/index')
          end
        end

      end
```

http://github.com/blazingcloud/rails_lessons/raw/master/2_controller/controllers/echo_controller_spec.rb

```
$ rake spec
uninitialized constant EchoController (NameError)

$ rails generate controller echo
      create  app/controllers/echo_controller.rb
      invoke  erb
      create    app/views/echo
      invoke  rspec
    conflict    spec/controllers/echo_controller_spec.rb
  Overwrite .../echo/spec/controllers/echo_controller_spec.rb? (enter "h" for
help) [Ynaqdh] n
        skip    spec/controllers/echo_controller_spec.rb
      invoke  helper
      create    app/helpers/echo_helper.rb
      invoke    rspec
      create      spec/helpers/echo_helper_spec.rb
```

Rails Generators

We've looked at how to use ActiveRecord, how to modify it with Ruby code and how it saves and retrieves data to and from a corresponding database table. We first created our Person model using the rails generate scaffold command, which generates the model, views and controller all at once. Calling:

```
$ rails generate scaffold person first_name:string last_name:string
```

is the same as calling

```
$ rails generate model person first_name:string last_name:string
$ rails generate controller people
```

and then creating the views manually.

It is so common for developers to generate model and controller, then hand-craft the views, that Rails provides an additional command that combines the above two commands into one:

```
$ rails generate resource person first_name:string last_name:string
```

More Typical Controller Example

```
$ rails new article_app

$ cd article_app
$ git init
$ git add .
$ git commit -m "new app"
```

edit Gemfile, at the bottom, uncomment the development/test group:

```
group :development, :test do
  gem 'rspec-rails'
end

$ bundle
$ rails generate rspec:install
$ rails generate rspec:controller articles
```

Copy the text from:

https://github.com/blazingcloud/rails_lessons/raw/master/2_controller/controllers/articles_controller_spec.rb

into /spec/controllers/articles_controller_spec.rb

```
$ rake spec
```

To make the spec pass you will need to create both an article model and a controller. Try using a resource generator (which take the same parameters as scaffold or a model generator)

REST

Rails includes a default implementation of the Model-View-Controller pattern with Representational State Transfer (REST). In 2000, Roy Fielding described REST as an architectural pattern well-suited to web applications. If you adhere to REST principles while designing your application, you will end up with a system that exploits the Web's architecture to your benefit.

Key REST principles:

- separation of concerns between UI and data storage
- stateless communication
- uniform interface between components

Rails implements this by creating a standard set of web pages that represent a database model as well as standard URLs, HTTP actions, and additional pages for accessing and modifying the model. Additionally Rails provides an easy mechanism for displaying different representations of a model, which we will discuss in more detail when we learn about controllers.

Note: Since many web clients, such as web browser forms and Flash, only support GET and POST, Rails has implement a common workaround, where you can specify a special parameter _method to specify put or delete and it will behave the same as if the true HTTP verb were used. You can see this using a tool like FireBug to inspect the POST parameters when using the web forms generated by scaffold to modify and delete objects.

Rails REST helpers

The generate scaffold command adds one like to routes.rb, which implements all of the REST URLs:

```
resources :people
```

You can inspect the result of this on the command-line using the useful rake task:

```
$ rake routes
people GET       /people(.:format)          {:action=>"index", :controller=>"people"}
people POST      /people(.:format)          {:action=>"create", :controller=>"people"}
new_person GET   /people/new(.:format)      {:action=>"new", :controller=>"people"}
edit_person GET  /people/:id/edit(.:format) {:action=>"edit", :controller=>"people"}
person GET       /people/:id(.:format)      {:action=>"show", :controller=>"people"}
person PUT       /people/:id(.:format)      {:action=>"update", :controller=>"people"}
person DELETE    /people/:id(.:format)      {:action=>"destroy", :controller=>"people"}
```

In addition to named routes, the view helper link_to uses a helpful utility url_for under the hood, allowing you to use models as resources and determine their corresponding URLs. You can see this in the rails console:

```
$ rails console
 >> bob = Person.first
 => #<Person id: 1, first_name: "Bob", last_name: "Hope", created_at: "2010-10-18 01:24:59",
updated_at: "2010-10-18 01:24:59">
>> app.url_for(bob)
 => "http://www.example.com/people/1"
>> app.people_url
=> "http://www.example.com/people"
>> app.person_url(bob)
=> "http://www.example.com/people/1"
>> app.person_url(1)
=> "http://www.example.com/people/1"
>> app.person_path(bob)
=> "/people/1"
```

Rails Default Implementation of REST

Index

The index action provides a list of all the resources of a specific type
Sample URL: http://localhost:3000/people
HTTP GET
Defualt web page: index.html

Show

The show action displays the details of a specific resources, by default identified by its ID
Sample URL: http://localhost:3000/people/42
HTTP GET
Defualt web page: show.html

New

The new action displays the a form for creating a new object
Sample URL: http://localhost:3000/people/new
HTTP GET
Defualt web page: new.html

Create

The create action accepts the form fields as parametrs to create a new model
Sample URL: http://localhost:3000/people
HTTP POST
no corresponding web page (re-directs to show.html to display the newly created object)

Edit

The edit action displays the a form for modifying a specific object
Sample URL: http://localhost:3000/people/42/edit
HTTP GET
Defualt web page: edit.html

Update

The update action accepts form fields as parametrs to modify the attributes of a model
Sample URL: http://localhost:3000/people/42
HTTP PUT
no corresponding web page (re-directs to show.html to display the modified object)

Delete

The delete action destroys a specific model
Sample URL: http://localhost:3000/people/42
HTTP DELETE
no corresponding web page (re-directs to index.html)

Form Helpers

Most of what you will do in Rails is using the resource pattern which is supported by the form_for helper. Let's look at a subset of the code in the form partial generated by scaffold (app/views/people/_form.html.erb)

```
<%= form_for(@person) do |f| %>

  <div class="field">
    <%= f.label :first_name %><br />
    <%= f.text_field :first_name %>
  </div>
  <div class="field">
    <%= f.label :last_name %><br />
    <%= f.text_field :last_name %>
  </div>
    <div class="field">
    <%= f.label :present %><br />
    <%= f.check_box :present %>
  </div>

  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

This will generate different code depending on whether @person was just created or fetched from the database. You can see this if you view source on the edit and new pages which both include this same partial code.

http://localhost:3000/people/new

```
<form accept-charset="UTF-8" action="/people"
        class="new_person" id="new_person" method="post">
```

http://localhost:3000/people/edit

```
<form accept-charset="UTF-8" action="/people/1"
        class="edit_person" id="edit_person_1" method="post">
```

Note that the action that is generated has a different URL. We can see how this resource-based URL generation works using the "url_for" helper in rails console:

```
Loading development environment (Rails 3.0.0)
>> fred = Person.new(:first_name => "Fred")
 => #<Person id: nil, first_name: "Fred", last_name: nil, created_at: nil,
updated_at: nil, present: nil>
>> fred.new_record?
 => true
>> app.url_for(fred)
 => "http://www.example.com/people"
>> wilma = Person.first
 => #<Person id: 1, first_name: "Wilma", last_name: "Flintstone", created_at:
"2010-10-21 19:39:29", updated_at: "2010-10-21 20:52:40", present: true>
>> wilma.new_record?
 => false
>> app.url_for(wilma)
 => "http://www.example.com/people/1"
```

Below all methods that are supported:

(most of text below from great article: http://code.alexreisner.com/articles/form-builders-in-rails.html)

```
# File: actionpack/lib/action_view/helpers/form_helper.rb
# Line: 75, in module ActionView::Helpers
module FormHelper
  def form_for(      record_or_name_or_array, *args, &proc)
  def fields_for(    record_or_name_or_array, *args, &block)
  def label(         object_name, method, text = nil, options = {})
  def text_field(    object_name, method, options = {})
  def password_field(object_name, method, options = {})
  def hidden_field(  object_name, method, options = {})
  def file_field(    object_name, method, options = {})
  def text_area(     object_name, method, options = {})
  def check_box(     object_name, method, options = {}, ...)
  def radio_button(  object_name, method, tag_value, options = {})
end
```

Error Messages

We learned with ActiveRecord that when a save fails (or valid? is called), all errors are stored for later access. So if we look at the roster example

```
Loading development environment (Rails 3.0.0)
>> p = Person.new
 => #<Person id: nil, first_name: nil, last_name: nil, created_at: nil,
updated_at: nil>
>> p.valid?
 => false
>> p.errors
 => #<OrderedHash {:last_name=>["can't be blank"], :first_name=>["can't be
blank"]}>
>> p.errors.full_messages
 => ["First name can't be blank", "Last name can't be blank"]
```

This is convenient for use in the view. Here's some example code from scaffold:

```
  <% if @person.errors.any? %>
      <div id="error_explanation">
            <h2><%= pluralize(@person.errors.count, "error") %>
                prohibited this person from being saved:</h2>
            <ul>
            <% @person.errors.full_messages.each do |msg| %>
                  <li><%= msg %></li>
            <% end %>
            </ul>
      </div>
<% end %>
```

Authenticity Token

The form helper also generates a div element with a hidden input for the "authenticity token." This is a security feature of Rails to protect from cross-site scripting attacks. It is generated for every form whose _method is not get.

```
<div style="margin:0;padding:0">
    <input name="authenticity_token" type="hidden"
           value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
</div>
```

You can turn this off (if you know what you are doing) in the Application controller, which by default looks like this:

app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery
end
```

ActiveRecord Associations

The Rails Guides has an excellent overview of associations:

http://guides.rubyonrails.org/association_basics.html

To get a feel for how associations work, we'll look at the most commonly used association is has_many. Before we dive in, let's make sure we are set up and understand our baseline. We'll use the "roster" app and create a new model for an association with "pets."

```
$ rails generate scaffold pets breed:string name:string person_id:integer

$ rails console
Loading development environment (Rails 3.0.0)
>> p = Person.create! :first_name => "Maria", :last_name => "Ortega"
=> #<Person id: 1, first_name: "Maria", last_name: "Ortega", present: nil,
created_at: "2009-11-15 13:49:14", updated_at: "2009-11-15 13:49:14">
>> p.pets
NoMethodError: undefined method `pets' for #<Person:0x17f1f38> from
/usr/local/lib/ruby/gems/1.8/gems/activerecord-2.3.4/lib/active_record/
attribute_methods.rb:260:in `method_missing'
      from (irb):4
```

Now edit your person model (app/models/person.rb) to declare the association:

```
class Person < ActiveRecord::Base
  has_many :pets
end

>> reload!
Reloading...
=> true

>> p = Person.create! :first_name => "Maria", :last_name => "Ortega"
=> #<Person id: 2, first_name: "Maria", last_name: "Ortega", present: nil,
created_at: "2009-11-15 13:51:07", updated_at: "2009-11-15 13:51:07">
>> p.pets
=> []
>> p.pets.build(:breed => "cat", :name => "fluffy")
=> #<Pet id: nil, breed: "cat", name: "fluffy", person_id: 2, created_at:
nil, updated_at: nil>
>> p.pets
=> [#<Pet id: nil, breed: "cat", name: "fluffy", person_id: 2, created_at:
nil, updated_at: nil>]
>> p.pets.length
=> 1
>> p.save!
=> true
```

```
>> p.pets
=> [#<Pet id: 1, breed: "cat", name: "fluffy", person_id: 2, created_at:
"2009-11-15 13:55:10", updated_at: "2009-11-15 13:55:10">]
>> p.pets.find_by_name("fluffy")
=> #<Pet id: 1, breed: "cat", name: "fluffy", person_id: 2, created_at:
"2009-11-15 13:55:10", updated_at: "2009-11-15 13:55:10">
>> p.pets[1] = Pet.new(:name => "Fred", :breed => "dog")
=> #<Pet id: nil, breed: "dog", name: "Fred", person_id: nil, created_at:
nil, updated_at: nil>
```

More Association Methods

Here are some more to try

```
p.pet_ids
p.pets.create
```

Note that "delete" methods always just affect the database; whereas, "destroy" methods call Ruby object destroy methods.

```
p.pets.delete_all
p.pets.destroy_all
```

The generated methods for associations are listed in the reference:

http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html