/Rails/build a blog with Rails 3.0

Part one In the first of a two-part tutorial, **Gavin Montague** explains how to take your first steps with Ruby on Rails 3.0 by building a generic blog

Knowledge needed	HTML, basic use of Command-Prompt, Terminal.app or Bash
Requires	Ruby, Rails, SQLite, browser, text editor
Project time	2-3 hours

Rails has become a favourite of devs looking for a powerful, agile way to build web apps quickly and cleanly. Version 3.0 marked a turning point when it merged with the more modular Merb. The result is a stronger, lighter framework that's suitable for projects of all sizes.

Here we're going to run through your first steps with Ruby on Rails. First we need the parts installed. This process varies between OSes, so pick your path below and then continue to the next section. We have four tools:

1. **Ruby** – The underlying language Rails is written in.
2. **Gem** – A package manager for managing and installing Ruby libraries.
3. **Rails** – Obviously.
4. **SQLite** – A lightweight SQL database.

Installation on Windows

Ruby's heritage is in Unix, but you can get a version for Windows by downloading the RubyInstaller from rubyinstaller.org. Go to **Downloads** and select **Ruby 1.8.7-pXXX**, where X is the patch number (302 at the time of writing). Install **ir**, making sure to check the option adding Ruby to your path. Next, open a prompt by selecting **Start Menu > Run > "cmd"** then type:

```
$ ruby –version
> ruby 1.8.7 (2010-08-16 patchlevel 302) [i386-mingw32]
```

If you don't see something similar to the screenshot below, see rubyinstaller.org for support. To get SQLite, download the administration tool (**sqlite-XXX.zip**) and the DLL (**sqlitedll-XXX.zip**) from www.sqlite.org/download.html. Copy the contents of both zips to **C:\Ruby\bin** then use **<gem>** to install Ruby support.

```
$ gem install sqlite3-ruby
```

Installation on OS X

Although both Ruby and Rails are included with more recent versions of OS X, they're too old for our purposes. We could individually install newer versions, but we'll use port to save time. Install port by following the instructions at www.macports.org/install.php with the pkg installer. Note that you'll also need the Apple Developer tools, which can be found on your OS X install disk or at developer.apple.com. With port installed, open a new Terminal (**/Application/Utilities/Terminal.app**) and run the following:

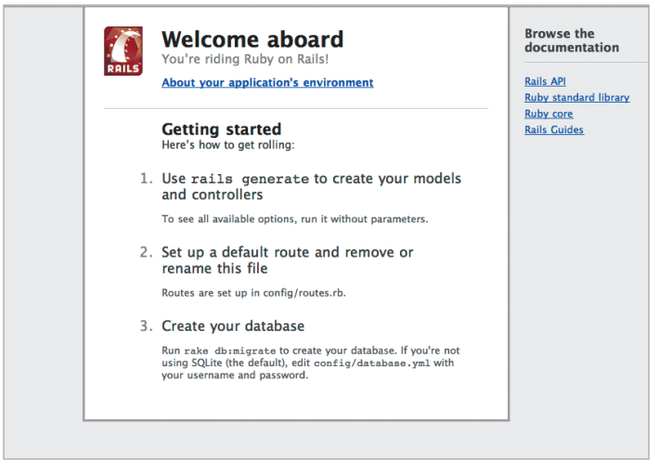
```
$ sudo port install ruby
$ sudo gem update –system
$ sudo gem uninstall rubygems-update
$ sudo gem install rails
$ sudo gem install sqlite3-ruby
```

Installation on Linux

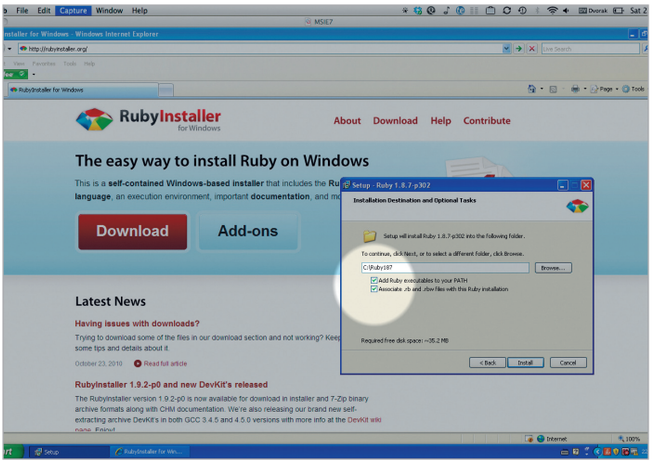
Ruby is very well supported by every Linux, but unfortunately the installation specifics vary between distributions. Your best bet is to check on the distribution's support forums for the correct path. Regardless of your operating system, you can now install Rails with the command:

```
$  gem install rails –version=3.0.1
```

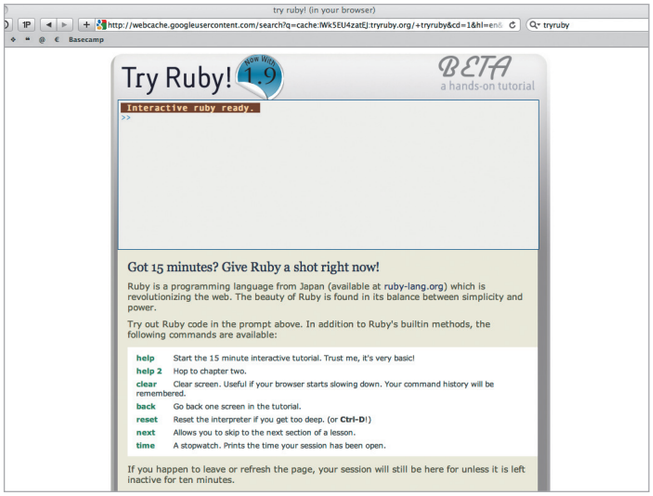
Keep your prompt open – we'll be using it a lot.



Start screen The default Ruby on Rails start screen – if you can't see this at **localhost:3000**, then something's gone horribly wrong!



Install Ruby The best way to get Ruby running on Windows is with rubyinstaller.org. Remember to add Ruby to your path during installation



In-browser tutorial If you'd like to try Ruby but can't install it on your machine, there's a comprehensive interactive tutorial at tryruby.org that runs in your browser

Ruby's syntax lends itself to very clear expression of ideas and a working grasp of its simple but powerful features can be picked up in a few hours. In technical terms, Ruby is a truly object-oriented language. This means that everything you interact with is treated as a self-contained 'box' of data that you perform operations on via **method**. For example:

```
a = 'a string"
a.reverse
=> "gnirts a"
array = [3, 2, 1]
array.sort
=> [1, 2, 3]
```

In Ruby you'll very likely spend most of your time working with classes, best thought of as a blueprint for creating objects. For example:

```
class Person
      attr_accessor :name
   def initialize(name)
         self.name = name
   end
   def greet
      "Hi, I'm #{name}"
   end
end
bob = Person.new "Bob"
bob.greet
=> "Hi, I'm Bob"
```

With Ruby, parenthesis are optional, and certain non-alphabetic characters can be used in function names, so it's perfectly valid to write:

```
name = "bob"
name.is_a? String
# => true
array = ["apple", "orange"]
array.empty?
#=> false
```

This makes Ruby ideal for writing very declarative, easily read code. Below is executable Rails code, but its meaning is quite apparent, even to non-coders:

```
before_save :set_timestamp, :unless=>:new_record?
```

One final point that often confuses newcomers is the use of meta-programming, or code-that-writes-code, as a standard tool. Rails makes quite heavy use of meta-programming internally to write methods as they're needed. With the **Article** class we'll shortly build I can search the database with either:

```
Article.find(:first, :conditions=>{:title=>"foo"})
```

or:

```
Article.find_by_title "foo"
```

Both methods do the same thing, but the trick is that the second one doesn't exist! Thanks to some meta-programming, Rails is able to take the more readable second version and turn it into the first longhand version as it's needed. Rails gives you a stack of these 'magic' methods, which help keep your code concise and readable but can occasionally lead to some hard-to-understand bugs. We'll be focusing on Rails from here on in, but I recommend you complete the tutorial at tryruby.org to get a feel for the language.

Our application

We're going to build a generic blog. You'll write articles and save them to a database. People will be able to browse them and, in part two, leave comments. Go to your prompt, navigate to your preferred working directory, with the **cd** command in everything else, and create your Rails app with the command:

```
$ cd ~/Desktop
$ rails blog
```

You'll see a bunch of output whizz by and a folder called **blog** appear, containing the outline of an app. Rails considers itself 'opinionated' – there are certain conventions about how you're expected to lay out your files and code, and woe betide those who don't. The advantage of this almost rabid devotion to convention is that the framework can make assumptions about your code: this leads to more reusable code and less effort. The first layer of this opinion is the layout of files in the project. A default Rails app contains 15 folders and files at the top level. Here, we'll only concern ourselves with **app**, **db** and **public**.

The App Folder

Rails adheres to the **Model/View/Controller** design pattern. This is an abstracted way of thinking about where responsibility for different behaviours lives. Rails maps these layers to three eponymous folders in **/app**.

Model: A meaningful collection of data. Users, messages, calendars and events are suitable model objects. It's up to the model to handle its own storage (maybe to a file or a database) and ensure that its internal state is both accurate and valid. In our blog, Articles and Comments will be models.

View: The view deals solely with presentation. For web apps it's generally the generation of HTML, but could be XML, JSON or PDF. We'll have views to display lists of articles, forms for editing them, and so forth.

>>

But it doesn't scale!

Exploding the biggest myth about RoR

One of the most persistent Rails myths is that it "doesn't scale". The usual evidence cited for this is Twitter. Its frequent downtime in 2008/9 was widely attributed to Rails not being able to support a high-traffic site. In truth, most of Twitter's problems lay in its architecture: what was really a messaging system had been built as a web app and that was causing growing pains. Twitter still uses Rails for its web-facing elements, but has re-engineered its underlying systems in more suitable languages.

In terms of web-app architecture, Rails actually scales pretty darn well. The LinkedIn Facebook app BumperSticker runs on Rails and serves many million page-views a month. The ecommerce juggernaut Groupon handles around nine million visits a month with Rails and the original Rails site, Basecamp, happily serves over 9,000 requests per minute.

In truth, scalability is never an easy problem to solve and most apps never reach the point where it becomes an issue. Rather than worrying about how your site will cope when it's getting a billion hits, leverage Rails to make building great apps easy and worry about scaling later.

>> **Controller:** The middleman – dealing with interpreting the incoming request and lining up the correct Models to pass to the View for display. In our app, the controller decides which Article is to be operated on, what the operation is and what templates to render.

The usefulness of this approach is that it creates clear areas of responsibility. For example, an Article will have certain requirements before it can be considered 'valid' (eg it must have a title and some text).

The controller will never know what the requirements are, but it will know how to ask an article object, 'Are you valid?'. Similarly, Articles knows nothing about HTML, but the View knows how to take the data from a Model and represent it as a form, table or XML.

```
$ rails generate scaffold article title:string body:text published_at:datetime
```

This generates a 'scaffold' – a boilerplate version of our MVC stack for the Article class. This scaffold contains just enough code to enable us to manipulate a database table called **articles** via a Model, a Controller and some Views.

An Article will have certain requirements before it can be considered 'valid'

Let's create the database table.

```
$  rake db:migrate
```

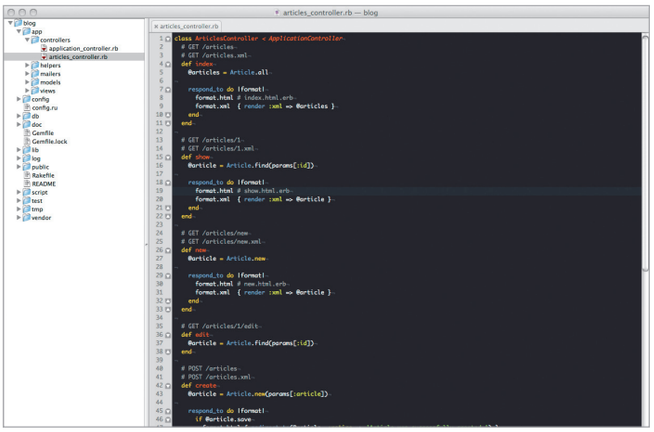
Rails has created an SQLite database and built a table to store our objects. It does this through migrations: timestamped Ruby files, which can be found in **/db/migrate**. Notice how we've not written anything that ties us to a kind of database. If I was running MySQL, PostgresSQL or SQLITE this migration would generate the correct SQL to build the table. By writing in Ruby rather than SQL we remain adaptable.

If you download a suitable SQLite management tool from www.sqlite.org/cvstrac/wiki?p=ManagementTools and open the **sqlite** file in **/db** you'll see we now have a table called **articles** containing the columns we specified above.

To see our scaffold in action, start the development web server. If you need to stop the server, press **Control+C**:

```
$ rails server
```

Open up your browser and visit http://localhost:3000. You'll be presented with the default Rails page – this is just the static index.html file in **/public** and can be ignored for the most part. Visit http://localhost:3000/articles and you'll be presented with an empty list. Click on **New** to create an article then try



Pure and simple Ruby is a very clear language – even without knowing Ruby most of the functions of the controller should be apparent

editing and deleting. The scaffold has given us enough code to manipulate the articles table.

On seeing the scaffold, most people react in one of two ways – "OMG! Rails writes code for me!" or "Oh, Rails is just a boilerplate generator". In fact, Rails is neither of these things. The scaffold is intended as a training wheel and a tool for bootstrapping development. As its name suggests, scaffolding gives support during construction: it's not meant to be permanent.

What the scaffold does give us is an idiomatic example of how to write code in Rails. We'll walk through that code now.

We'll start with the controller. Open **app/controllers/articles_controller.rb** in your editor.

Visiting http://localhost:3000/article/1/edit will create an **articles_controller** and run its **edit** action, passing in the parameter **{:id=>1}**. The output from this method is sent back to the browser. Don't worry about how this happens for now: Rails routing is very customisable, but the default behaviour is fine for us. Let's look at some of our controller methods.

Show

```
def show
  @article = Article.find(params[:id])
  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @article }
  end
end
```

Good design in Rails generally equates to lots of small methods doing one thing, and this method is no exception. We first try to fetch an Article with a primary key matching the **id** passed to us. Data passed through from the browser is made available via the **params** object (if you've ever used PHP, **params** is basically analogous to **$_REQUEST**). Notice that we have an **@ prefix** on the article variable – this makes it into an instance variable, and available in the template.

Next, we have the **responds_to** block. This concerns itself with deciding what to output back to the user. If the user has asked for HTML we don't have to specify any behaviour; by convention Rails will render the template in **/app/views/articles/show**. Alternatively, the scaffold sets can output our article as XML. Try going back to your browser and appending **.xml** to the URL. You may have to view-source, but you should see an XML version of the **article** object. This can be very convenient for passing data between web servers and releasing APIs for your application.

The template for our **show** action is very basic – it simply prints out the various fields of our **Article** object.

```
<p><b>Title:</b><%= @article.title %></p>
```

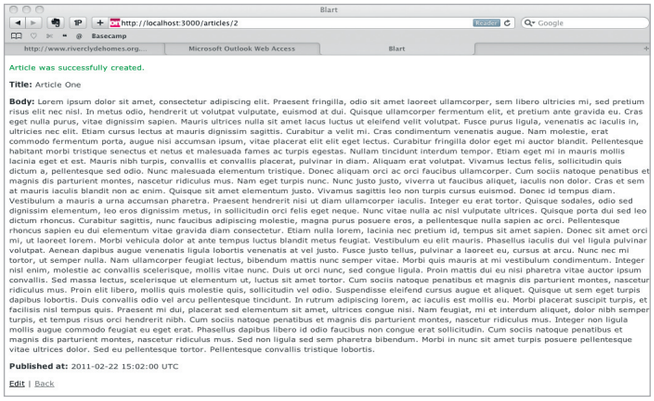
There are various ways of writing templates in Rails, but the default is erb: embedded Ruby. Syntactically, this is closely related to PHP in that the templates actually contain running Ruby code. Although this makes it possible to write 'real' code in the template, the views should only concern themselves with the logic about how the data is presented: anything smarter than that is up to the Controller and Model.

Edit

Now let's look at the process for editing and updating our articles. The **edit** method is even more trivial than **show** – all the controller has to do is ask the Model to fetch the correct row from the database and then pass it onto the **edit.html.erb** template, again through the magic of convention.

So what is our **edit** template supposed to do? Well, it should output a form representing the Article we retrieved in the controller. Rails provides a library for this and it integrates very neatly with our **ActiveRecord** based models. Regardless of whether the user is creating a new Article or editing an existing one, the form will look the same. The scaffold has extracted the common part of the page to a separate template called **_form.html.erb**. In Rails parlance this is a partial – a template used by other templates.

The form partial is more complex than our **show** action, but it's still quite readable. Try viewing-source on the rendered page in your browser and working through the lines of code in the template to see how the form gets built.



The scaffold It's not the prettiest of things, but the Rails **scaffold** command will give you an easy foothold in getting your application up and running

Update

Our **edit** method is the most complex one in the scaffold, but it's still a bit short.

```
def update
  @article = Article.find(params[:id])
  respond_to do |format|
    if @article.update_attributes(params[:article])
      format.html { redirect_to(@article, :notice => 'Article was successfully updated.') }
else
      format.html { render :action => "edit" }
end
  end
end
```

The method begins with us finding the object we want to update, as with **show** and **edit**, but then we have a little bit of logic. The **params** object contains all the data passed from our form in a Hash (like an array, but with non-numeric keys). We can pass this Hash directly to our Article, which replaces its title, body etc and then attempts to save to the database. If the object is saved we call **redirect_to** to send the browser to a different URL (we'll look at how Rails knows where to redirect to in part two). If the save fails we need to explicitly tell Rails to render the **edit** template because the user needs to know something went wrong. But what could possibly go wrong?

Models

Now let's look at the class we've been manipulating all this time. Considering that an Article knows what fields it has, can read/write to a database and search its table, it's probably quite a big file. Open **/app/models/article.rb**:

```
class Article < ActiveRecord::Base
end
```

Where's all this behaviour coming from, then?

The top line of this file is our class definition – it says that Article inherits from the **ActiveRecord::Base** class.

Inheritance is too big a topic to cover here, but it's fundamental to object-oriented Programming.

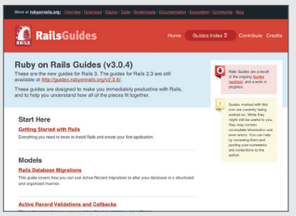
In essence it allows methods defined in our parent, or superclass, to be used, or inherited, in the child class. **ActiveRecord** knows how to talk to the database, work out what table maps to what class and do all the heavy lifting of SQL so our Articles get the ability for free.

Notice that we don't even have to tell it what table it should read/write with and what fields in should have – Rails intelligently maps the **Article** class to the **articles** table in the DB. It then goes on to determine that our article should have a title, body etc because that's what columns we have in the DB.

We said that Articles ought to require a title before they can be saved. We can implement this by asking the Model to validate one or more conditions. Add the following inside your **Article** class, then go back to your browser and try to create an Article with no title or an overly long one.

Resources Where to learn more

Programming Ruby
pragprog.com/titles/ruby/programming-ruby
Deals with 1.9, which has some differences from 3, but nothing to cause problems for a learner.



RailsGuides
guides.rubyonrails.org
To delve deeper into Rails, the best sources of written information are the community-maintained guides available from the Rails website.

Railscasts
railscasts.com
Ryan Bates hosts a catalogue of almost 250 five- to 10-minute videos, suitable for all levels.

Rails API
apidock.com/rails
The full API is available on the official Rails site, but I prefer the version at apidock.com.

```
class Article < ActiveRecord::Base
  validates :title,
      :presence => true,
      :length => {:maximum => 25}
end
```

This adds a rule: in order to be valid, a title must be present and can't be more than 25 characters. With the validation callback we say, 'Make sure all our validation conditions are mean – if they aren't then don't save the object'.

validates is an example of a callback function. **ActiveRecord** defines several points in the object lifecycle where we can ask for code to be run without having to get down into the guts. Say we wanted to add a list of all the times an article had been edited. Add the following inside the Article class.

```
before_save :append_timestamp, :unless=>:new_record?
def append_timestamp
self.body << "\nEdited at #{Time.now}"
end
```

Each time you save your Article it will call **append_timestamp** as one of its **before_save** callbacks.

This has been a very quick introduction to Rails. In part two next issue we'll dive a bit more deeply, using Rails to manage the relationship between Comments and Articles. We'll also see how we can leverage the huge number of third-party plug-ins and gems to add features to your application.

About the author

Name Gavin Montague
Site leftbrained.co.uk
Areas of expertise Rails, usability.
Clients BBC, News International, Dell, National Trust for Scotland, itison.com
Ideal dinner party guest Scout Brandie. That's not funny, but he promised to stop annoying me if he got a mention