/Rails/build a blog with Rails 3.0

Download the files! > All the files you need for this tutorial can be found at netm.ag/rails-215

Part two In the second of a two-part tutorial, **Gavin Montague** explains how to extend your blog by integrating third-party libraries for extra functionality

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

In this tutorial, we'll extend the blog from part one and dive deeper into the Rails framework. We'll also look at integrating third-party libraries to give extra functionality for very little effort.

If you missed part one in issue 214, you can download a PDF, along with the tutorial files for both parts, from netm.ag/rails-215. You'll also need Ruby, Rails and SQLite installed.

You'll have to make use of the Rails, Rake and Gemtools, so we'll start with a quick recap of how to use the command line.

To open a prompt on Windows, select Start Menu > Run > "cmd". This should bring up a dos-prompt.

On OS X, open the Terminal.app from /Applications/Utilities.

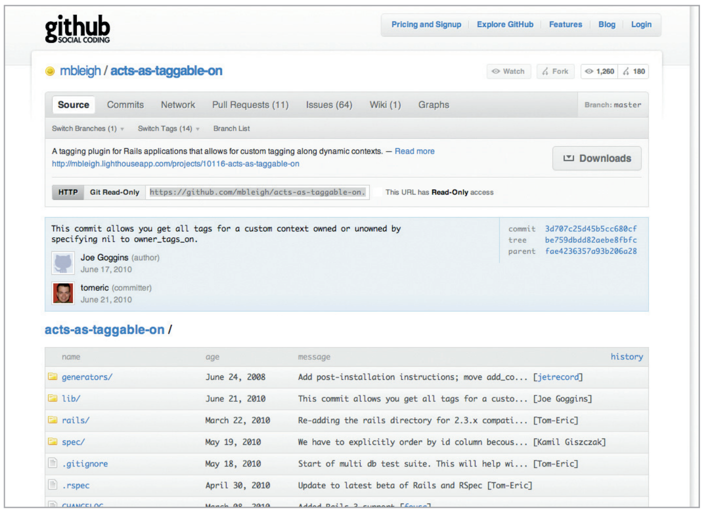
On Linux you'll usually find a shell program somewhere in your distro's menus.

With the prompt, you can navigate to your Rails project by using the **cd** command to move between directories:

```
$ cd .                                        # move up a level
$ cd ./foo                                    # move into a subdirectory called foo
$ cd                                          # return to your home directory
```

Once in place, you can start Rails' web server by using the following command and then visiting http://localhost:3000 in your browser:

```
$ rails server
```



mbleigh / acts-as-taggable-on

A tagging plugin for Rails applications that allows for custom tagging along dynamic contexts. — Read more
http://mbleigh.lighthouseapp.com/projects/10116-acts-as-taggable-on

https://github.com/mbleigh/acts-as-taggable-on. This URL has **Read-Only** access

```
This commit allows you get all tags for a custom context owned or unowned by
specifying nil to owner_tags_on.
```

Joe Goggins (author) June 17, 2010
tomeric (committer) June 21, 2010

```
commit  3d707c25d45b5cc680cf
tree    be759dbdd82aebe8fbfc
parent  fae4236357a93b206a28
```

acts-as-taggable-on /

name	age	message	history
generators/	June 24, 2008	Add post-installation instructions; move add_co... [jetrecord]	
lib/	June 21, 2010	This commit allows you get all tags for a custo... [Joe Goggins]	
rails/	March 22, 2010	Re-adding the rails directory for 2.3.x compati... [Tom-Eric]	
spec/	May 19, 2010	We have to explicitly order by id column becous... [Kamil Giszczak]	
.gitignore	May 18, 2010	Start of multi db test suite. This will help wi... [Tom-Eric]	
.rspec	April 30, 2010	Update to latest beta of Rails and RSpec [Tom-Eric]	

Gem of an idea The **acts-as-taggable-on** gem will add the ability to tag your objects and more – see github.com/mbleigh/acts-as-taggable-on for full details

If you want to interact with the constituent objects of your app, invoke the console:

```
$ rails console
> article = Article.new
> article.title = "Hello World"
> article.body = "Some words"
> article.save
=> true
> found_article = Article.find_by_title "Hello World"
=> #<Article id: 1, title: "Hello World", ...>
```

Visitor comments

The next step in our blog is to enable readers to leave comments. Let's think about what this will involve in terms of the MVC (**Model**, **View**, **Controller**) pattern from part one.

At the **Model** level, we'll need a way of storing or retrieving comments from our database. Additionally, we'll need to describe the relationship between an article and its comments and vice versa. This will be familiar to you if you've done any database stuff as a **1-to-N** or a **has-many** relationship.

In our **Controller**, we'll want to be able to parse user input into instances of our **Comment** model and try to save them. Finally, in the **View**, we want to both present these comments and provide a form that will enable visitors to bestow their collective wisdom on us. We previously used the **scaffold** command as a shortcut to the article **Model**, **Controller** and **Views**, but this time we'll walk through each individual step of the stack in a bit more detail. We'll start at the bottom by defining our **Model** class:

```
$ rails generate model Comment name:string body:text article:references
```

This will stub out a class in **app/models/comment.rb** for us, and define a new migration in **/db/migrate** that we can use to create the database table via **rake**:

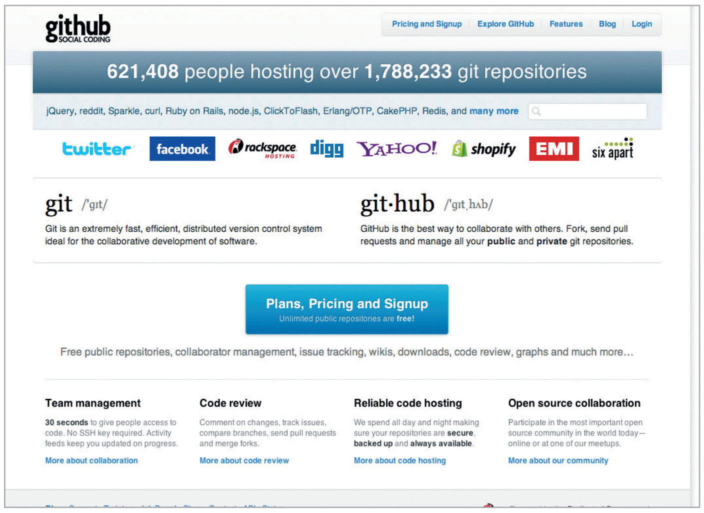
```
$ rake db:migrate
```

We can now start a console and manipulate comment. As with our articles, Rails' **ActiveRecord** framework handles all the database interaction for us:

```
$ rails console
> comment = Comment.new
=> #<Comment id: nil, name: nil, body: nil....>
> comment.name = "Bob"
> comment.body="Hello world"
> comment.save
> Comment.find(:first)
=> #<Comment id: 1, name: "Bob", body: "Hello world", ....>
```

Our comments aren't much use on their own, though; we need to define them by the article they belong to. Add the following lines into your article and comment classes in **app/models**:

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



github SOCIAL CODING

Pricing and Signup | Explore GitHub | Features | Blog | Login

621,408 people hosting over 1,788,233 git repositories

jQuery, reddit, Sparkle, curl, Ruby on Rails, node.js, ClickToFlash, Erlang/OTP, CakePHP, Redis, and many more

git /'gɪt/

Git is an extremely fast, efficient, distributed version control system ideal for the collaborative development of software.

git·hub /'gɪtˌhʌb/

GitHub is the best way to collaborate with others. Fork, send pull requests and manage all your **public** and **private** git repositories.

Plans, Pricing and Signup
Unlimited public repositories are **free!**

Free public repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs and much more…

Team management
30 seconds to give people access to code. No SSH key required. Activity feeds keep you updated on progress.
More about collaboration

Code review
Comment on changes, track issues, compare branches, send pull requests and merge forks.
More about code review

Reliable code hosting
We spend all day and night making sure your repositories are **secure**, **backed up** and **always available**.
More about code hosting

Open source collaboration
Participate in the most important open source community in the world today—online or at one of our meetups.
More about our community

Code repository As well as hosting the source code for Ruby and Rails, Github is the home for most gems, plug-ins and open source projects written in Ruby

```
   has_many :comments, :order=>"id"
end
class Comment < ActiveRecord::Base
   belongs_to :article
end
```

I've said previously that Ruby and Rails combine to form a very expressive syntax, and this is another good example: the meaning of the code shines through:

```
$ rails console
> article = Article.first
=> #<Article id: 1, title: "An article about rails"....>
> article.comments
=> []
> article.comments.create :name=>"Adam", :body=>"Hiya, Saturn"
=> #<Comment id: 2, name: "Adam", body: "Hiya, Saturn" ...>
> article.comments.create :name=>"Zoe", :body=>"Ciao, Mars"
=> #<Comment id: 3, name: "Zoe", body: "Ciao, Mars", article_id: 1 ....>
> article.comments.count
=> 2
> article.comments.find_by_name("Zoe")
=> #<Comment id: 3, name: "Zoe", body: "Ciao, Mars", article_id: 1 ....>
> comment = Comment.last
=> #<Comment id: 3, name: "Zoe", body: "Ciao, Mars", ...>
> comment.article
=> #<Article id: 1, title: "one", body: "two"...>
```

We now have access to our articles and comments not just on their own, but also by the relationship between them. We can tell an article to add a comment, which will automatically be associated with the correct article at the database level. We can also ask a comment which article it belongs to, and so on.

Visitor comments

We're only touching on it here, but Rails provides an incredibly rich abstraction layer that does most of the SQL heavy lifting. It's rare to have to resort to raw SQL in Rails; you have the option to if necessary, but it's rarely required. Developers are left to concentrate on what makes their application different from everyone else's, rather than having to reinvent the wheel every time they search the database or write a record.

Create a few more comments for your articles on the console, then move on to displaying them in your app. Open up views/articles/show.html.erb and append the following:

```
<h3>Comments</h3>
<%= render @article.comments %>
```

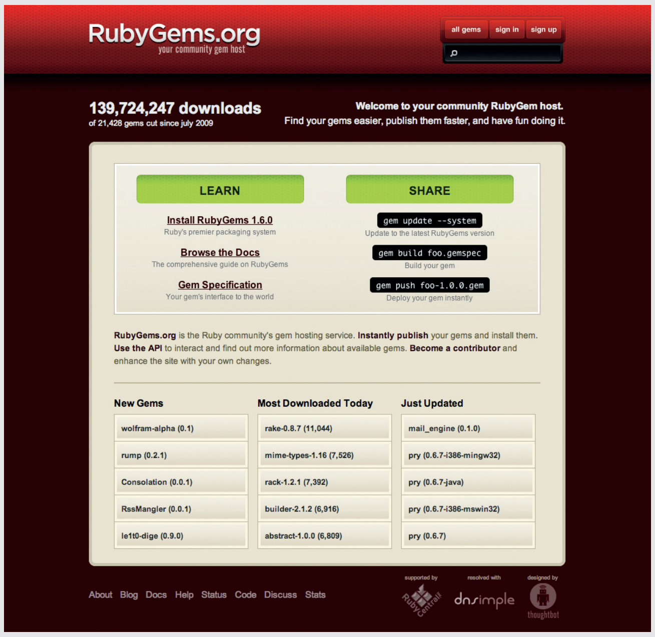
GitHub and RubyGems

Has your problem already been solved?

The **gem** command works by interrogating a number of online sources to discover Gems. The most important is rubygems.org, the most authoritative list of published gems, and a good port of call if you're looking to see if anyone else has already solved your problem.

If rubygems.org is the library for Ruby, GitHub (github.com) is its playground. As well as hosting the source code for Ruby and Rails (along with jQuery, YUI, Erlang, Prototype and more) it's the home for most gems, plug-ins and open source projects written in Ruby.

Open a free account and you'll be able to explore the source code for projects and even fork them to make your own customisations. Oh, and did I mention that it's all written in Rails?



RubyGems.org
your community gem host

all gems | sign in | sign up

139,724,247 downloads
of 21,428 gems cut since july 2009

Welcome to your community RubyGem host.
Find your gems easier, publish them faster, and have fun doing it.

LEARN	SHARE
Install RubyGems 1.6.0 Ruby's premier packaging system	`gem update --system` Update to the latest RubyGems version
Browse the Docs The comprehensive guide on RubyGems	`gem build foo.gemspec` Build your gem
Gem Specification Your gem's interface to the world	`gem push foo-1.0.0.gem` Deploy your gem instantly

RubyGems.org is the Ruby community's gem hosting service. **Instantly publish** your gems and install them. **Use the API** to interact and find out more information about available gems. **Become a contributor** and enhance the site with your own changes.

New Gems	Most Downloaded Today	Just Updated
wolfram-alpha (0.1)	rake-0.8.7 (11,044)	mail_engine (0.1.0)
rump (0.2.1)	mime-types-1.16 (7,526)	pry (0.6.7-i386-mingw32)
Consolation (0.0.1)	rack-1.2.1 (7,392)	pry (0.6.7-java)
RssMangler (0.0.1)	builder-2.1.2 (6,916)	pry (0.6.7-i386-mswin32)
le1t0-dige (0.9.0)	abstract-1.0.0 (6,809)	pry (0.6.7)

About Blog Docs Help Status Code Discuss Stats

supported by | resolved with | designed by

List of gems RubyGems.org is the most authoritative list of published gems – a good place to find out if someone else already has a solution to your problem

Start the server and visit one of your articles – **Error: "Missing partial comments/comment.** Ah, we've asked Rails to render another template inside our show one, so we best create it. Add a new file at **app/views/comments/_comment.html.erb**.

A template that isn't rendered directly but called from inside other 'action' templates is referred to as a 'partial', and always begins with an underscore:

```
<h3><%= comment.name %> says...</h3>
<%= simple_format(comment.body) %>
<hr />
```

You should now get a better result.

You might be feeling a bit dizzy now: how did Rails know what to do? We didn't tell it what template to render; we didn't tell it to loop through our comments; and where did the variable comment come from in our partial? Well, one of Rails' core principles is 'Convention over Code'. If we follow the best practices of Rails then it can accurately infer a great deal about our intent that would otherwise clutter our code.

We've already used Convention over Code in our models. By naming our tables in a certain way, we were able to use **has_many** and **belongs_to** to describe the relationship between our tables. We passed an array to the render method, so Rails inferred we wanted to loop through it and render the template once for each item in it. We didn't give an explicit template to render, so Rails assumed that we wanted to render a certain path based on the type of object in our array. In this case, it decided we wanted to use the

>>

>> template at **comments/_comment.html.erb**. All this implied meaning can seem like magic at first, and you're free to skip this shorthand and be more explicit in your code. I could have rewritten the above as:

```
<%= render :partial=>"/comments/comment", :collection=>@article.
comments %>
```

Or even:

```
<% @article.comments.each do |comment| %>
  <%= render :partial=>"/comments/comment", :object=>comment %>
<% end %>
```

However, the original version is more concise and, once you know the rules, clearer in its meaning.

Now let's give our users the facility to add comments. To do this, we'll attach a form to the bottom of our article. First, though, take a look at the **config/routes.rb** file.

Routing in Rails can be a bit confusing if you've previously only worked with static HTML sites, or raw PHP (where there's a one-to-one mapping between a URL and file). If you visit **/example/page.html** then that's what loads. Not so with Rails. We have a step between the incoming request and the Controller that deals with it. This gives Rails lots of flexibility in how we map URLs to controllers and their actions. We'll keep it simple and just let our app know how it should handle comments.

Routing can be a bit confusing if you've only worked with static HTML sites or raw PHP

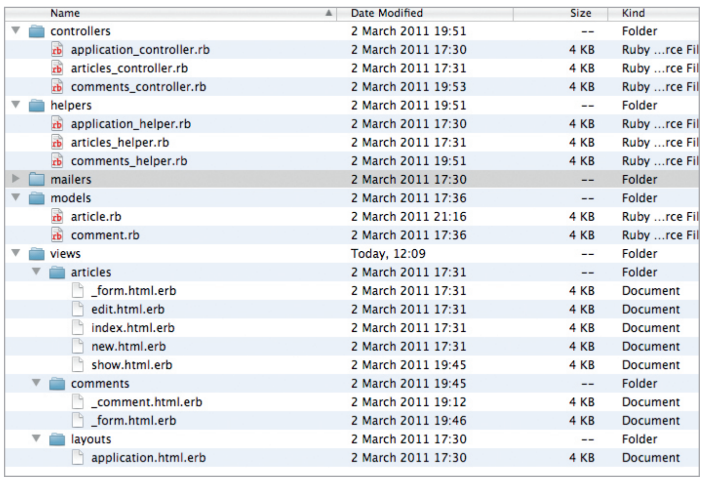
Open **config/routes.rb** and replace the second line with:

```
resources :articles do |article|
resources :comments, :only=>[:create]
end
```

On the command line, run **rake** to see which URLs your app now responds to:

```
$ rake routes
article_comments POST /articles/:article_id/comments(.:format)
{:action=>"create", :controller=>"comments"}
```

That's nice: Rails tells us what our URL will look like, which parameters we need and what our action should be called.



Name	Date Modified	Size	Kind
controllers	2 March 2011 19:51	--	Folder
application_controller.rb	2 March 2011 17:30	4 KB	Ruby ...rce Fil
articles_controller.rb	2 March 2011 17:31	4 KB	Ruby ...rce Fil
comments_controller.rb	2 March 2011 19:53	4 KB	Ruby ...rce Fil
helpers	2 March 2011 19:51	--	Folder
application_helper.rb	2 March 2011 17:30	4 KB	Ruby ...rce Fil
articles_helper.rb	2 March 2011 17:31	4 KB	Ruby ...rce Fil
comments_helper.rb	2 March 2011 19:51	4 KB	Ruby ...rce Fil
mailers	2 March 2011 17:30	--	Folder
models	2 March 2011 17:36	--	Folder
article.rb	2 March 2011 21:16	4 KB	Ruby ...rce Fil
comment.rb	2 March 2011 17:36	4 KB	Ruby ...rce Fil
views	Today, 12:09	--	Folder
articles	2 March 2011 17:31	--	Folder
_form.html.erb	2 March 2011 17:31	4 KB	Document
edit.html.erb	2 March 2011 17:31	4 KB	Document
index.html.erb	2 March 2011 17:31	4 KB	Document
new.html.erb	2 March 2011 17:31	4 KB	Document
show.html.erb	2 March 2011 19:45	4 KB	Document
comments	2 March 2011 19:45	--	Folder
_comment.html.erb	2 March 2011 19:12	4 KB	Document
_form.html.erb	2 March 2011 19:46	4 KB	Document
layouts	2 March 2011 17:30	--	Folder
application.html.erb	2 March 2011 17:30	4 KB	Document

A place for everything Rails is highly prescriptive as to how your app is laid out, but the enforced structure pays dividends

Back in **articles/show.html.erb**, add:

```
<h2>Add a Comment</h2>
<%= render :partial=>"/comments/form" %>
```

Then create the partial to render the form in the comments folder **app/views/comments/_form.html.erb**:

```
<% form_for [@article, @comment||Comment.new] do |form| %>
<p><%= form.label :name %>: <%= form.text_field :name %></p>
<p><%= form.label :body %>: <%= form.text_area :body %></p>
<%= submit_tag "Comment" %>
<% end %>
```

Again, there's a fair amount of Convention over Code here. Rails is working out which URL the form should submit to from the objects we're passing in. It also provides methods for wrapping the values of our **comment** object in form fields.

In Rails, forms and inputs become transparently easy to populate. You'll now be able to see the form, but submitting it will throw in an error because we haven't added the controller and action to deal with the request:

```
$ rails generate controller comments
```

Open your new **controllers/comments_controllers.rb** file and add:

```
def create
@article = Article.find(params[:article_id])
@comment = @article.comments.build(params[:comment])
respond_to do |format|
if @comment.save
format.html { redirect_to(@article, :notice => 'Comment was successfully
added.') }
else
format.html { render :action => "new" }
end
end
end
```

Our method is now in place to translate form values (held in params) into Ruby objects and save the new comment to the database before taking the user back to the article. Notice that there's a branch in the code for when the comment doesn't save and we want to render the template 'new' with the form in it. If you're feeling adventurous, try completing this path (hint: start by telling your comment model that it needs a name before it can be saved). See the section on validation in part one for more details.

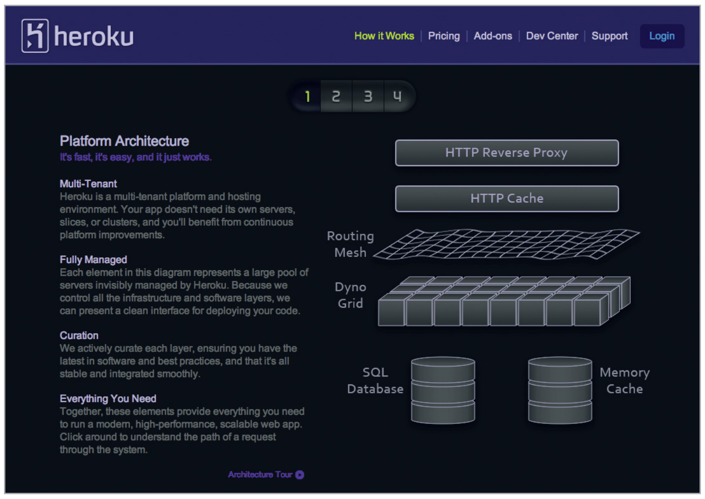
In essence, we've now added a complete feature to our application. While we could tweak the exact details of it, perhaps requiring a valid email address, or adding permalinks to individual comments, the overall structure wouldn't change: **Models** are marshalled via a **Controller** and the output of their interactions is display in a **View**. Armed with this, you can start adding other features to your blog.

Using Gems

A blog wouldn't be a blog without tagging. It's a scientific fact, which I've just made up, that humanity was unable to find any information until the invention of the Tag-Cloud at CERN in 2001 by a team of 12 French physicists and a Belgian who everyone assumed was French.

It's pretty boring to build a tagging system, though, and it's actually a very common problem. If we substitute out our domain object (the article) then we get a statement that could apply to recipes, music tracks, bookmarks or anything else. So that we can browse Things by topic, each Thing should be able to take an arbitrary number of text labels. We should then be able to browse by these labels.

If the problem isn't unique to our domain, let's see if we can get someone else to solve it for us. Ruby has a rich ecosystem of libraries that are distributed as self-contained packages called Gems and managed by a command called, funnily enough, **gem**. Rails itself is distributed as a Gem. You can see this with the command:



Platform Architecture
It's fast, it's easy, and it just works.

Multi-Tenant
Heroku is a multi-tenant platform and hosting environment. Your app doesn't need its own servers, slices, or clusters, and you'll benefit from continuous platform improvements.

Fully Managed
Each element in this diagram represents a large pool of servers invisibly managed by Heroku. Because we control all the infrastructure and software layers, we can present a clean interface for deploying your code.

Curation
We actively curate each layer, ensuring you have the latest in software and best practices, and that it's all stable and integrated smoothly.

Everything You Need
Together, these elements provide everything you need to run a modern, high-performance, scalable web app. Click around to understand the path of a request through the system.

Free account Ruby cloud platform Heroku Rails (heroku.com) is an easy way to go about hosting your Rails website – plus it's free to sign up and get started

```
$ gem list
```

You'll be presented with a list of all the libraries you've installed. One of the key features of **gem** is its dependency management. When you installed Rails, **gem** automatically included all the other libraries it depends on. Also notice that all the gems have their version numbers listed; you can use this feature to jump between different versions of the same gem in your projects as required. That's an advanced topic, though. Let's go back to our tagging example:

```
$ gem install acts-as-taggable-on
```

After running this command, you should see output indicating that the gem has been successfully installed. We need to tell our app to load it by editing **Gemfile**. This acts as an index telling our app which gems to load when it starts, and where to find them if they're not installed:

```
gem 'acts-as-taggable-on'
```

Next up we're going to need some tables to hold our tags and their relationships; **acts-as-taggable-on** can do this for us:

```
$ rails generate acts_as_taggable_on:migration
$ rake db:migrate
```

You'll see that this has added some tables to your database. We just have to let the app know what we want to be taggable:

```
class Article < ActiveRecord::Base
acts_as_taggable_on :tags
```

That's all the code you need to make your articles taggable. Fire up a console and explore:

```
> a = Article.first
=> #<Article id: 1, title: "one", body: "two", ...>
> b = Article.last
=> #<Article id: 2, title: "Second", body: "lorem", ...>
> a.tag_list = "Red, Green, Blue"
=> "Red, Green, Blue"
> b.tag_list = "Pink, Blue"
=> "Pink, Blue"
> a.save
> b.save
> a.tags
=> [#<ActsAsTaggableOn::Tag id: 1, name: "Red">,
#<ActsAsTaggableOn::Tag id: 2, name: "Blue">,
#<ActsAsTaggableOn::Tag id: 3, name: "Green">]
> b.tags
=> [#<ActsAsTaggableOn::Tag id: 2, name: "Blue">,
#<ActsAsTaggableOn::Tag id: 4, name: "Pink">]
> Article.tagged_with "Blue"
=> [#<Article id: 1, title: "one", body: "two", ...>, #<Article id: 2,
title: "Second", ...>]
> Article.tagged_with "Red"
=> [#<Article id: 1, title: "one", ...>]
> a.find_related_tags
=> [#<Article id: 2, title: "Second", ...>]
```

Going further

The **acts-as-taggable-on** gem can do a lot more than this: find out exactly what from the project homepage (github.com/mbleigh/acts-as-taggable-on). Why not work through the readme on GitHub and see if you can extend your **Article** form to add tags, and then have your **show** action provide links to related items?

Gems – and their close relative, plug-ins – give Rails an extremely rich system of extensible features. You can quickly add everything from swear-word filters to full ecommerce systems to your Rails applications, leveraging other developers' experience to build your site better and faster.

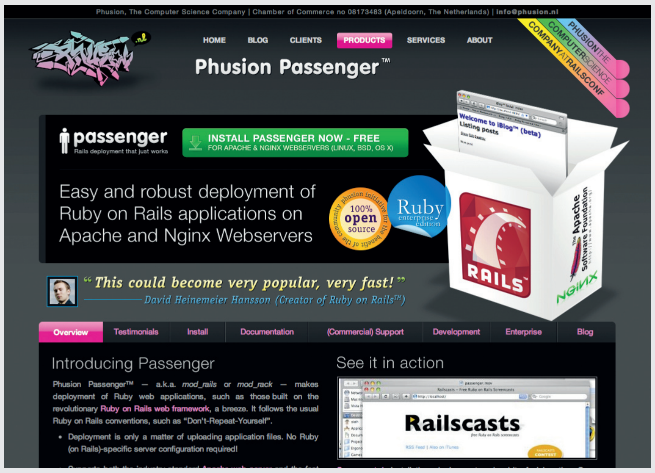
That concludes our whistlestop tour of Rails 3.0, and we've only scratched the surface of the aid it offers to developers. I hope your interest has been piqued and you explore what the framework can do and see what other features you can add to your blog. ●

Hosting Rails

How to get your project up and running

Hosting a Rails website can require a bit more thought than a static website, or a PHP one, but it's not hard. Phusion's Passenger (modrails.com) adds Rails support to the world's most popular webserver, Apache. If your hosting company doesn't already provide Passenger support, there are plenty of others that do.

For example, Heroku: Rails hosts as a cloud service (heroku.com). Not only does it provide an incredibly neat way of deploying your app with almost no effort; you can also host a development account for free.



Phusion, The Computer Science Company | Chamber of Commerce no 08173483 (Apeldoorn, The Netherlands) | info@phusion.nl

Phusion Passenger™

Easy and robust deployment of Ruby on Rails applications on Apache and Nginx Webservers

"This could become very popular, very fast!"
David Heinemeier Hansson (Creator of Ruby on Rails™)

Introducing Passenger

Phusion Passenger™ — a.k.a. *mod_rails* or *mod_rack* — makes deployment of Ruby web applications, such as those built on the revolutionary Ruby on Rails web framework, a breeze. It follows the usual Ruby on Rails conventions, such as "Don't-Repeat-Yourself".

- Deployment is only a matter of uploading application files. No Ruby (on Rails)-specific server configuration required!

See it in action

Get on board Most good hosting firms will provide support for Passenger, which offers easy deployment of Ruby applications on Apache servers

About the author

Name	Gavin Montague
Site	leftbrained.co.uk
Twitter	@gavinmontague
Areas of expertise	Rails, usability
Clients	BBC, News International, Dell, National Trust for Scotland, itison.com
Favourite comedy show	Collings & Herrin