

Rails 4 Quickly

Bala Paranj

About Author

Bala Paranj has a Master's degree in Electrical Engineering from The Wichita State University. He has over 15 years of experience in the software industry. He was a Java developer before he found Ruby in May 2006. He has also developed iPhone and iPad apps.

His academic background includes mathematics and electronics. Some of his clients include telecommunication, banks, financial and electronic design automation companies. He consults through his company Zepho Inc for clients within USA. He loves to learn and teach technical topics. He has spoken at Silicon Valley Ruby Meetup, San Francisco Ruby Meetup and Silicon Valley Code Camp. You can reach him at bala.paranj@zepho.com with any feedback.

About Reader

This book assumes that you have already installed Ruby 2.0, Rails 4 and your favorite IDE. The reader must already have a basic understanding of Ruby language. This is a short book. The objective is to bring you upto speed in Rails 4 quickly. Hence the title Rails 4 Quickly. This book is written for beginners who want to learn the fundamentals. It will give you solid foundation for you to build upon.

The main focus is on Rails. You will not find any discussion on Cucumber, Git, Heroku, RSpec, FactoryGirl or any other irrelevant topics. It is written in a practical and hands-on approach to learning Rails. You learn by doing so you will benefit the most by following the instructions as you read each chapter.

How to Read this Book

This book is written in a step-by-step hands-on tutorial style. You must read and follow the instructions to work through the application we will be developing. It is written to be read sequentially. You must also make an attempt to do the exercises. This will make your brain more receptive to absorbing the concepts.

Software Versions Used

Ruby Gems : 2.1.5 Ruby : 2.0 Rails : 4.0

Source Code

Source code is available from bitbucket Git repo : <https://bitbucket.org/bparanj/rails-4-blog>

Acknowledgments

This book is the result of teaching Rails tutorials from Silicon Valley Ruby meetup. The members of Silicon Valley Ruby meetup provided me early feedback on every chapter. This book is also an experimentation on applying Lean Startup principles to self publishing. The advice that was very powerful to me was ‘Do not develop a product in a vacuum’.

I owe a debt to the Ruby creator Matz for creating such a beautiful language and the Ruby community for creating useful frameworks and gems to make developers life easy. I hope this book makes your learning process a little easy.

Table of Contents

1. Running the Server
2. Hello Rails
3. Model
4. Model View Controller
5. View to Model
6. Update Article
7. Show Article
8. Delete Article
9. View Duplication
10. Relationships
11. Delete Comment
12. Restricting Operations

Appendix

- A. Self Learning

1. Running the Server

Objective

- To run your rails application on your machine.

Steps

Step 1

Check the versions of installed ruby, rails and ruby gems by running the following commands in the terminal:

```
$ ruby -v
ruby 2.0.0p247 (2013-06-27 revision 41674) [x86_64-darwin12.5.0]
```

```
$ rails -v
Rails 4.0.0
```

```
$ gem env
RUBYGEMS VERSION: 2.1.5
```

Step 2

Change directory to where you want to work on new projects.

```
$ cd projects
```

Step 3

Create a new Rails project called blog by running the following command.

```
$ rails new blog
```

Step 4

Open a terminal and change directory to the blog project.

```
$ cd blog
```

Step 5

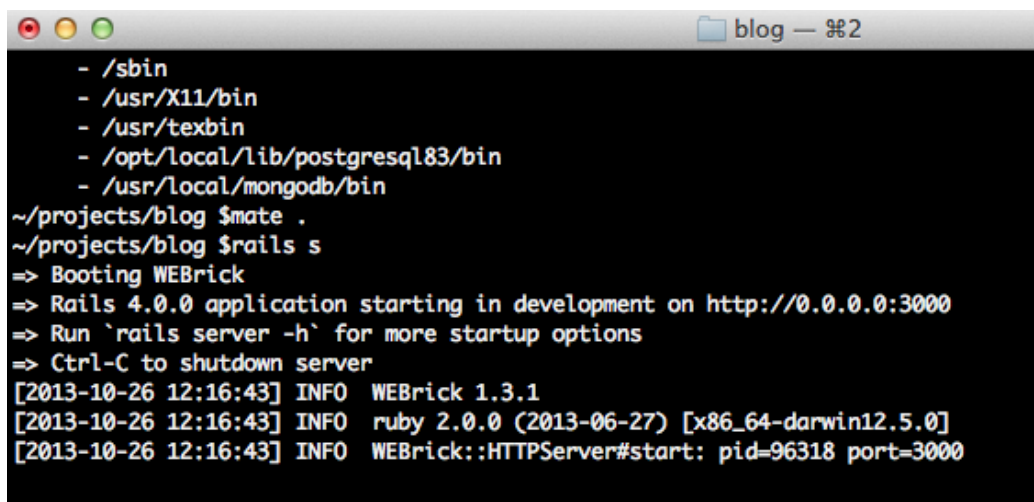
Open the blog project in your favorite IDE. For textmate :

```
$ mate .
```

Step 6

Run the rails server:

```
$ rails s
```

A screenshot of a terminal window titled "blog — 2". The terminal shows the following output:

```
- /sbin
- /usr/X11/bin
- /usr/texbin
- /opt/local/lib/postgresql83/bin
- /usr/local/mongodb/bin
~/projects/blog $mate .
~/projects/blog $rails s
=> Booting WEBrick
=> Rails 4.0.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
[2013-10-26 12:16:43] INFO  WEBrick 1.3.1
[2013-10-26 12:16:43] INFO  ruby 2.0.0 (2013-06-27) [x86_64-darwin12.5.0]
[2013-10-26 12:16:43] INFO  WEBrick::HTTPServer#start: pid=96318 port=3000
```

Figure 1: Rails Server

Step 7

Open a browser window and enter `http://localhost:3000`

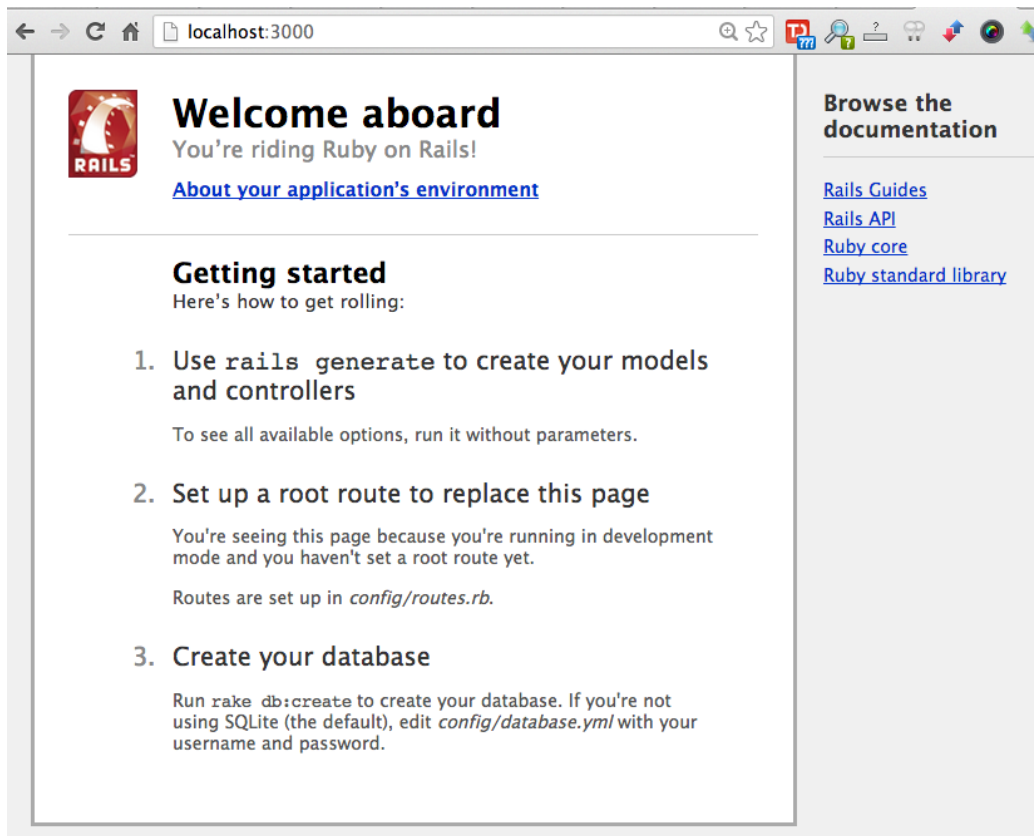


Figure 2: Welcome Aboard

Step 8


You can shutdown your server by pressing Control+C. If you use Control+Z, you will send the process to the background which means it will still be running but the terminal will be available for you to enter other commands. If you want to see the server running to see the log messages you can do :

```
$ fg
```

which will bring the background process to the foreground.

Step 9

Click on the 'About' link and check the versions of software installed. If the background of the about section is yellow, installation is fine. If it is red then something is wrong with the installation.



Welcome aboard

You're riding Ruby on Rails!

[About your application's environment](#)

Ruby version	2.0.0 (x86_64-darwin12.5.0)
RubyGems version	2.1.5
Rack version	1.5
Rails version	4.0.0
JavaScript Runtime	JavaScriptCore
Active Record version	4.0.0
Action Pack version	4.0.0
Action Mailer version	4.0.0
Active Support version	4.0.0

ActionDispatch::Static
Rack::Lock

<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x0000010397fb4
Rack::Runtime
Rack::MethodOverride
ActionDispatch::RequestId
Rails::Rack::Logger
ActionDispatch::ShowExceptions

Figure 3: About Environment

Explanation

The rails generator automatically runs the Bundler command `bundle` to install your application dependencies by reading the Gemfile. The Gemfile contains all the gems that your application needs. `rails s` (s is a short-cut for server) runs your server on your machine on port 3000.

Summary

In this lesson you learned how to run the server locally. We also saw how to check if everything is installed properly on our machine. In the next lesson you will learn how to create a home page for your web application.

2. Hello Rails

Objective

- To create a home page for your web application.

Steps

Step 1

Open the config/routes.rb file in your IDE, routes.rb defines the routes that is installed on your web application. Rails will recognize the routes you define in this configuration file.

Step 2

Look for the line :

```
# root 'welcome#index'
```

Step 3

Uncomment that line by removing #.

```
root 'welcome#index'
```

The method root() takes a string parameter. In this case it maps the home page of your site to welcome controller (class), index action (method).

Step 4

Go to the terminal and change directory to the blog project and run:

```
rake routes
```

```
~/projects/blog $rake routes
Prefix•Verb•URI Pattern•Controller#Action
root•GET•/•welcome#index•
~/projects/blog $
```

Figure 4: Rake Output

The output of this command shows you the installed routes. Rails will be able to recognize the GET request for welcome page.

The output has four columns, namely Prefix, Verb, URI Pattern and Controller#Action.

Prefix is the name of the helper that you can use in your view and controller to take the user to a given view or controller. In this case it is `root_path` or `root_url` that is mapped to your home page.

Verb is the Http Verb such as GET, POST, PUT, DELETE etc.

URI Pattern is what you see in the browser URL. In this case, it is `www.example.com`

Step 5

Go to the browser and reload the page : `http://localhost:3000`

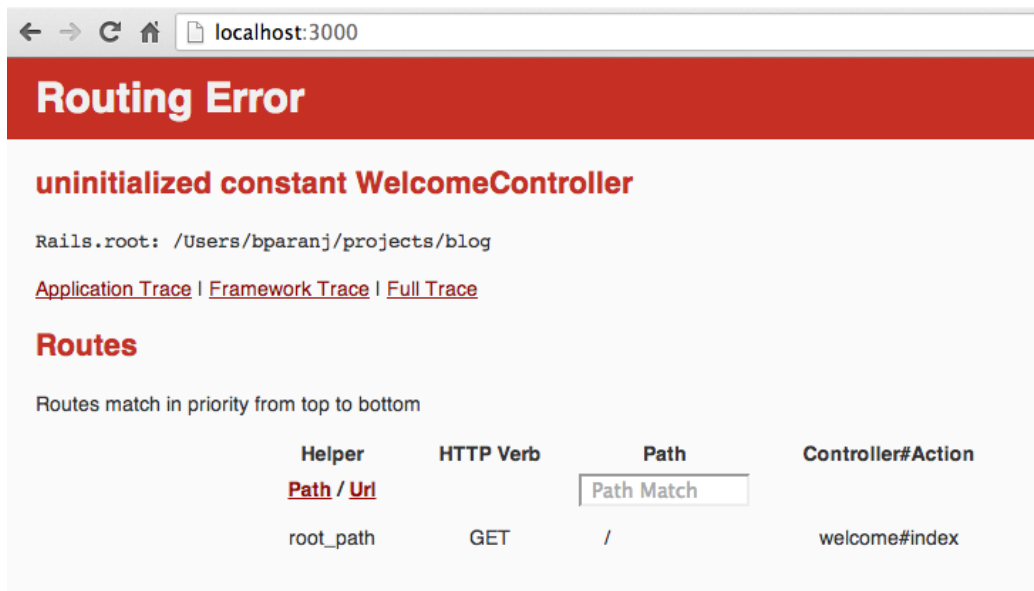


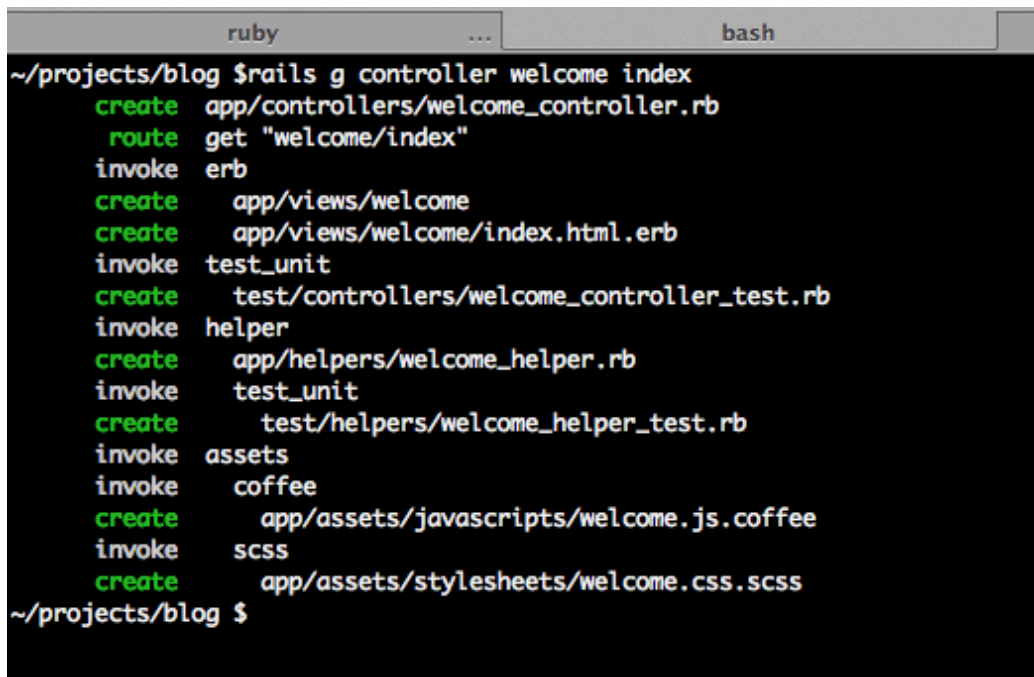
Figure 5: Create Controller

We see the uninitialized constant WelcomeController error. This happens because we don't have a welcome controller.

Step 6

Go the root of the project and type:

```
$ rails g controller welcome index
```

A terminal window with a dark background and light-colored text. The window has two tabs at the top: 'ruby' and 'bash'. The prompt is '~/projects/blog \$'. The command entered is 'rails g controller welcome index'. The output shows a series of actions performed by Rails, with 'create' and 'route' commands in green and 'invoke' commands in white. The actions include creating a controller, routes, views, tests, helpers, assets, and coffee/scss files.

```
~/projects/blog $ rails g controller welcome index
  create  app/controllers/welcome_controller.rb
  route   get "welcome/index"
  invoke  erb
  create  app/views/welcome
  create  app/views/welcome/index.html.erb
  invoke  test_unit
  create  test/controllers/welcome_controller_test.rb
  invoke  helper
  create  app/helpers/welcome_helper.rb
  invoke  test_unit
  create  test/helpers/welcome_helper_test.rb
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/welcome.js.coffee
  invoke  scss
  create  app/assets/stylesheets/welcome.css.scss
~/projects/blog $
```

Figure 6: Create Controller

rails command takes the arguments g for generate, then the controller name and the action.

Step 7

Reload the web browser again.

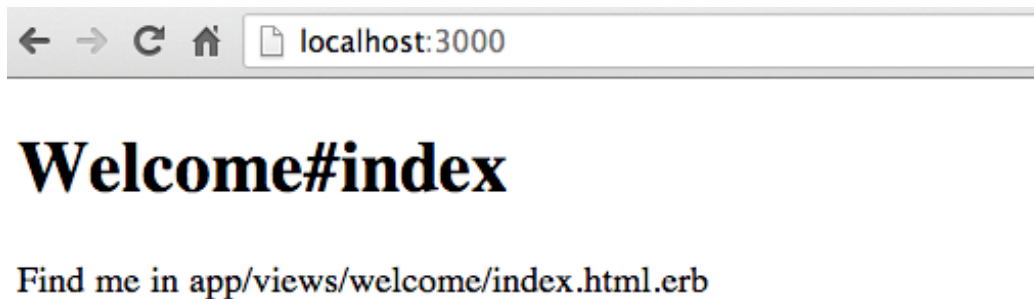


Figure 7: Welcome Page

You will now see the above page.

Step 8

Go to `app/views/index.html.erb` and change it to 'Hello Rails' like this:

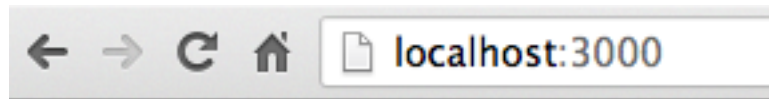
```
<h1>Hello Rails</h1>
```

Save the file.

You can embed ruby in `.html.erb` files. In this case we have html only. We will see how to embed ruby in views in the next lesson.

Step 9

Reload the browser.



Hello Rails

Figure 8: Hello Rails

Now you will see 'Hello Rails'.

Step 10

Open the `welcome_controller.rb` in `app/controllers` directory and look at the `index` action.

Step 11

Look at the terminal where you have the rails server running, you will see the request shown in the following image:

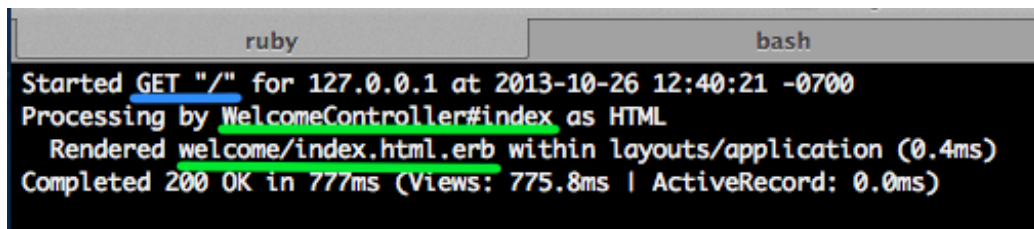


Figure 9: Server Output

You can see that the browser made a GET request for the resource `/` which is the home page of your site. The request was processed by the server where Rails recognized the request and it routed the request to the welcome controller index action. Since we did not do anything in the index action, Rails looks for the view that has the same name as the action and renders that view. In this case, it is `app/views/welcome/index.html.erb`.

Exercise

Can you go to `http://localhost:3000/welcome/index` and explain why you see the contents shown in the page?

Before you go to the next page and read the answer, make an attempt to answer this question.

Answer : You will see the same ‘Hello Rails’ page. Because if you check the rails server log you can see it made a request : GET ‘/welcome/index’ and if you look at the routes.rb file, you see :

```
get "welcome/index"
```

This definition is used by the Rails router to handle this request. It knows the URI pattern of the format ‘welcome/index’ with http verb GET must be handled by the welcome controller index action.

Delete the get “welcome/index” line in the routes.rb file. Reload the page : <http://localhost:3000/welcome/index>.

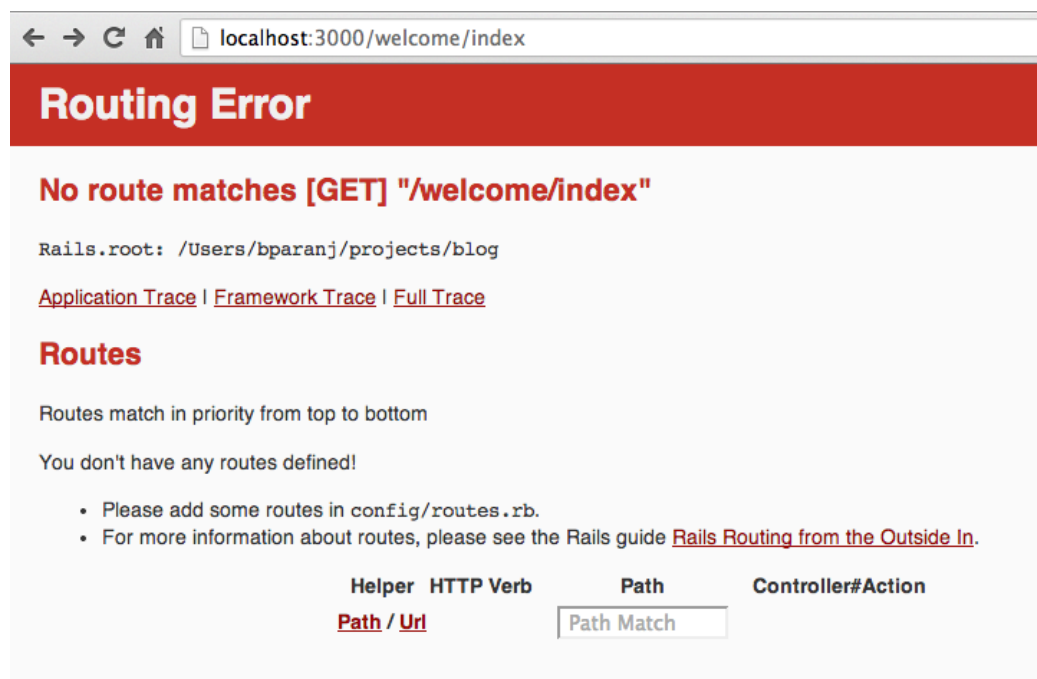


Figure 10: Welcome Index

You will now see the error page.

Summary

In this lesson we wrote a simple Hello Rails program. We saw how the view and controller work in Rails to handle browser requests. We have seen just

the VC part of MVC framework. We will see how the model fits in the MVC framework in the next lesson.

3. Model

Objective

- To learn the model part M of the MVC framework

Steps

Step 1

Open config/routes.rb file and add :

```
resources :articles
```

Save the file. Your file should like this :

```
Blog::Application.routes.draw do
  root 'welcome#index'

  resources :articles
end
```

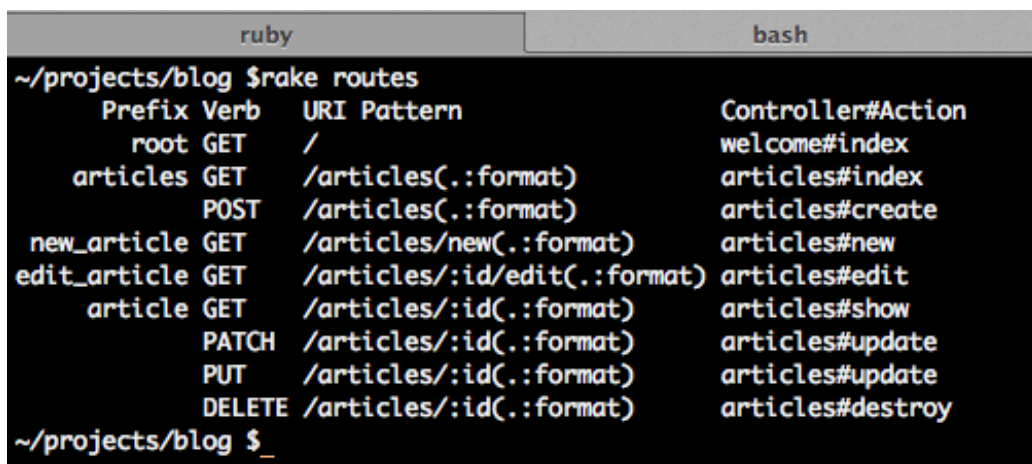
What is a resource? Resource can represent any concept. For instance if you read the documentation for [Twitter API](#), you will see that Timeline is a resource. It is defined in the documentation as collections of Tweets, ordered with the most recent first. There may not be a one-to-one correspondence between a resource and database table. In our case we have one-to-one correspondence between the database table articles and the article resource.

We have a plural resource so we will have index page that displays a list of all the articles in our case. Singular resource can be used when you don't need index action, for instance if a customer has a billing profile then from the perspective of a customer you can use a singular resource for `billing_profile`. From an admin perspective you could have a plural resource to manage billing profiles of customers (most likely using admin namespace in the routes).

Step 2

Go to the blog directory in the terminal and run:

```
$ rake routes
```



```
~/projects/blog $rake routes
      Prefix Verb   URI Pattern               Controller#Action
    root GET    /                       welcome#index
  articles GET    /articles(.:format)      articles#index
           POST   /articles(.:format)      articles#create
 new_article GET    /articles/new(.:format)  articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
  article GET    /articles/:id(.:format)  articles#show
           PATCH  /articles/:id(.:format)  articles#update
           PUT    /articles/:id(.:format)  articles#update
           DELETE /articles/:id(.:format)  articles#destroy
~/projects/blog $_
```

Figure 11: Installed Routes

The output shows that defining the articles resource in the routes.rb gives us routing for :

Action	Purpose
create	creating a new article
update	updating a given article
delete	deleting a given article
show	displaying a given article
index	displaying a list of articles

Since we have plural resources in the routes.rb, we get the index action. If you had used a singular resource :

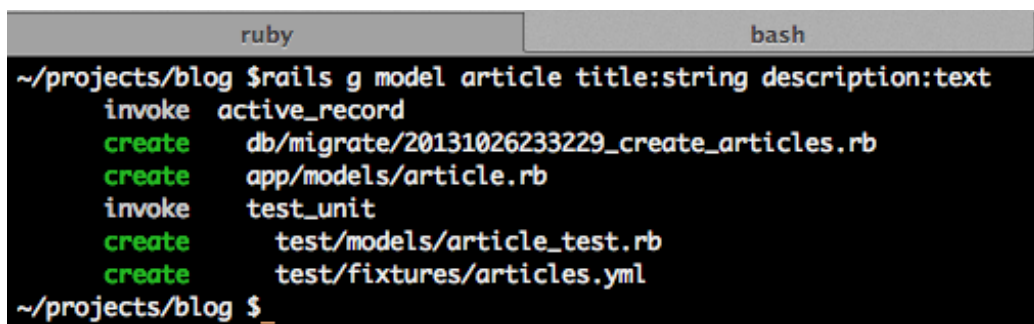
```
resource :article
```

then you will not have a routing for index action. Based on the requirements you will choose a singular or plural resources for your application.

Step 3

In the previous lesson we saw how the controller and view work together. Now let's look at the model. Create an `active_record` object by running the following command:

```
$ rails g model article title:string description:text
```



```
ruby bash
~/projects/blog $ rails g model article title:string description:text
  invoke  active_record
  create  db/migrate/20131026233229_create_articles.rb
  create  app/models/article.rb
  invoke  test_unit
  create  test/models/article_test.rb
  create  test/fixtures/articles.yml
~/projects/blog $ _
```

Figure 12: Article Model

In this command the rails generator generates a model by the name of article. The `active_record` is the singular form, the database will be plural form called as articles. The articles table will have a title column of type string and description column of type text.

Step 4

Open the file `db/migrate/xyz_create_articles.rb` file. The `xyz` will be a timestamp and it will differ based on when you ran the command.

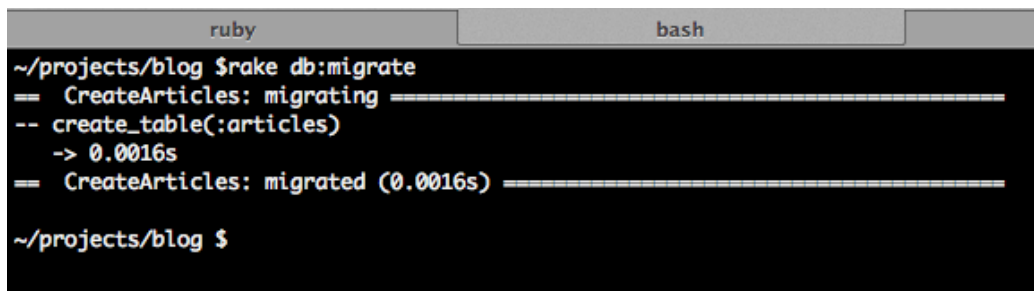
There is a `change()` method in the migration file. Inside the `change()` method there is `create_table()` method that takes the name of the table to create and also the columns and it's data type.

In our case we are creating the articles table. Timestamps gives created_at and updated_at timestamps that tracks when a given record was created and updated respectively. By convention the primary key of the table is id. So you don't see it explicitly in the migration file.

Step 5

Go to the blog directory in the terminal and run :

```
$ rake db:migrate
```

A terminal window with two tabs, 'ruby' and 'bash', both showing the same content. The terminal is in the directory ~/projects/blog. It shows the command 'rake db:migrate' being executed. The output indicates that the 'CreateArticles' migration is running, specifically the 'create_table(:articles)' step, which takes 0.0016s to complete. The final output shows 'CreateArticles: migrated (0.0016s)' and the prompt returns to '~/.projects/blog \$'.

```
~/projects/blog $rake db:migrate
== CreateArticles: migrating =====
-- create_table(:articles)
   -> 0.0016s
== CreateArticles: migrated (0.0016s) =====

~/projects/blog $
```

Figure 13: Create Table

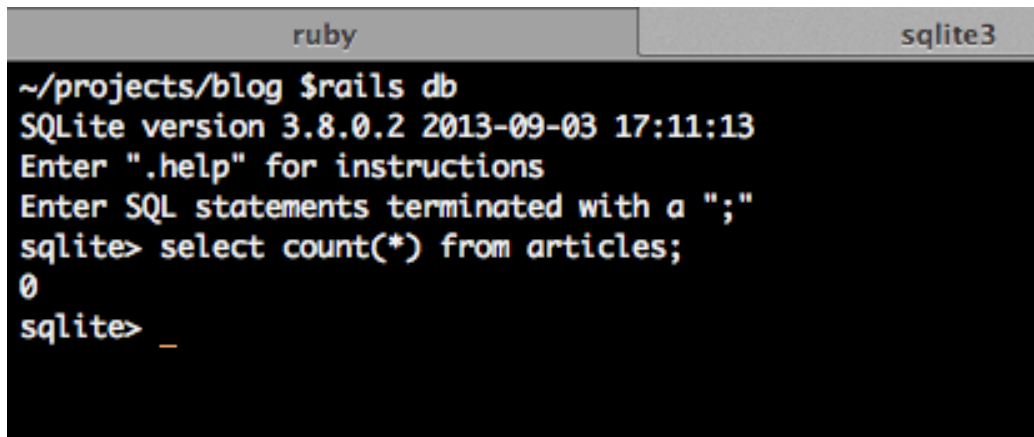
This will create the articles table.

Step 6

In the blog directory run:

```
$ rails db
```

This will drop you into the database console. You can run SQL commands to query the development database.



```
~/projects/blog $rails db
SQLite version 3.8.0.2 2013-09-03 17:11:13
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select count(*) from articles;
0
sqlite> _
```

Figure 14: Rails Db Console

Step 7

In the database console run:

```
select * from articles;
```

You can see from the output there are no records in the database.

Step 8

Open another tab in the terminal and go to the blog directory. Run the following command:

```
$ rails c
```

c is the alias for console. This will take you to rails console where you can execute Ruby code and experiment to learn Rails.

Step 9

Type :

`Article.count`

in the rails console.

ruby	sqlite3
<pre>~/projects/blog \$rails c Loading development environment (Rails 4.0.0) 2.0.0p247 :001 > Article.count (0.1ms) SELECT COUNT(*) FROM "articles" => 0 2.0.0p247 :002 ></pre>	

Figure 15: Rails Console

You will see the count is 0. Let's create a row in the articles table.

Step 10

Type :

`Article.create(title: 'test', description: 'first row')`

ruby	sqlite3	ruby
<pre>2.0.0p247 :003 > Article.create(title: 'test', description: 'first row') (0.1ms) begin transaction SQL (4.9ms) INSERT INTO "articles" ("created_at", "description", "title", "updated_at") VALUES (?, ?, ?, ?) [["creat ed_at", Sun, 27 Oct 2013 01:17:59 UTC +00:00], ["description", "first row"], ["title", "test"], ["updated_at", Sun, 27 O ct 2013 01:17:59 UTC +00:00]] (3.0ms) commit transaction => #<Article id: 1, title: "test", description: "first row", created_at: "2013-10-27 01:17:59", updated_at: "2013-10-27 01:17:59"> 2.0.0p247 :004 ></pre>		

Figure 16: Create a Record

The Article class method create creates a row in the database. You can see the ActiveRecord generated SQL query in the output.

Exercise 1

Check the number of articles count by using the database console or the rails console.

Step 11

Let's create another record by running the following command in the rails console:

```
$ article = Article.new(title: 'record two', description: 'second row')
```

ruby	sqlite3	ruby
<pre>2.0.0p247 :007 > article = Article.new(title: 'another record', description: 'different way to create row') => #<Article id: nil, title: "another record", description: "different way to create row", created_at: nil, updated_at: nil> 2.0.0p247 :008 ></pre>		

Figure 17: Article Instance

Now it's time for the second exercise.

Exercise 2

Check the number of articles count by using the database console or the rails console. How many rows do you see in the articles table? Why?

The reason you see only one record in the database is that creating an instance of Article does not create a record in the database. The article instance in this case is still in memory.

```
ruby      sqlite3      ruby
2.0.0p247 :007 > article = Article.new(title: 'another record', description: 'different way to create row')
=> #<Article id: nil, title: "another record", description: "different way to create row", created_at: nil, updated_at: nil>
2.0.0p247 :008 > Article.count
(0.6ms) SELECT COUNT(*) FROM "articles"
=> 1
2.0.0p247 :009 > _
```

Figure 18: Article Count

In order to save this instance to the articles table, you need to call the save method like this:

```
$ article.save
```

```
2.0.0p247 :009 > article.save
(0.1ms) begin transaction
SQL (0.8ms) INSERT INTO "articles" ("created_at", "description", "title", "updated_at") VALUES (?, ?, ?, ?) [["created_at", Sun, 27 Oct 2013 01:31:51 UTC +00:00], ["description", "different way to create row"], ["title", "another record"], ["updated_at", Sun, 27 Oct 2013 01:31:51 UTC +00:00]]
(1.4ms) commit transaction
=> true
2.0.0p247 :010 > _
```

Figure 19: Saving a Record

Now query the articles table to get the number of records. We now have some records in the database. In the next chapter we will display all the records in articles table on the browser.

Summary

In this chapter we focused on learning the model part M of the MVC framework. We experimented in the rails console and database console to create records in the database. In the next lesson we will see how the different parts of the MVC interact to create database driven dynamic web application.

4. Model View Controller

Objectives

- Learn how the View communicates with Controller
- Learn how Controller interacts with the Model and how Controller picks the next View to show to the user.

Context

Why MVC architecture? The advantage of MVC is the clean separation of View from the Model and Controller. It allows you to allocate work to teams according to their strengths. The View layer can be developed in parallel by the front-end developers without waiting for the Model and Controller parts to be completed by the back-end developers.

If we agree on the contract between the front-end and back-end by defining the data representation exchanged between the client and server then we can develop in parallel.

Steps

Step 1

Let's modify the existing static page in `welcome/index.html.erb` to use a view helper for hyperlink:

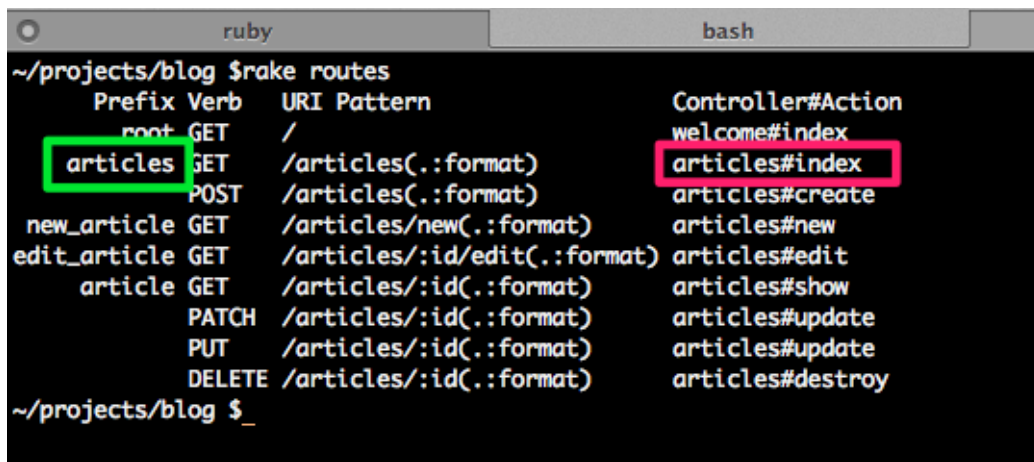
```
<%= link_to 'My Blog', ? %>
```

The tag `<%=` should be used whenever you want the generated output to be shown in the browser. If it not to be shown to the browser and it is only for dynamic embedding of Ruby code then you should use `<% %>` tags.

The `link_to(text, url)` method is a view helper that will generate an html hyperlink that users can click to navigate to a web page. In this case we want the user to go to articles controller index page. Because we

want to get all the articles from the database and display them in the `app/views/articles/index.html.erb` page.

So the question is what should replace the `?` in the second parameter to the `link_to` view helper? Since we know we need to go to articles controller index action, let use the output of rake routes to find the name of the view_helper we can use.



```
~/projects/blog $rake routes
      Prefix Verb   URI Pattern               Controller#Action
root GET    /                   welcome#index
articles GET    /articles(.:format)      articles#index
          POST   /articles(.:format)      articles#create
new_article GET    /articles/new(.:format)  articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
article GET    /articles/:id(.:format)  articles#show
          PATCH  /articles/:id(.:format)  articles#update
          PUT    /articles/:id(.:format)  articles#update
          DELETE /articles/:id(.:format)  articles#destroy
~/projects/blog $
```

Figure 20: Rake Routes

As you can see from the output, for `articles#index` the Prefix value is `articles`. So we can use either `articles_path` (relative url) or `articles_url` (absolute url).

Step 2

Change the link as follows :

```
<%= link_to 'My Blog', articles_path %>
```

Step 3

Go to the home page by going to the `http://localhost:3000` in the browser.

What do you see in the home page?

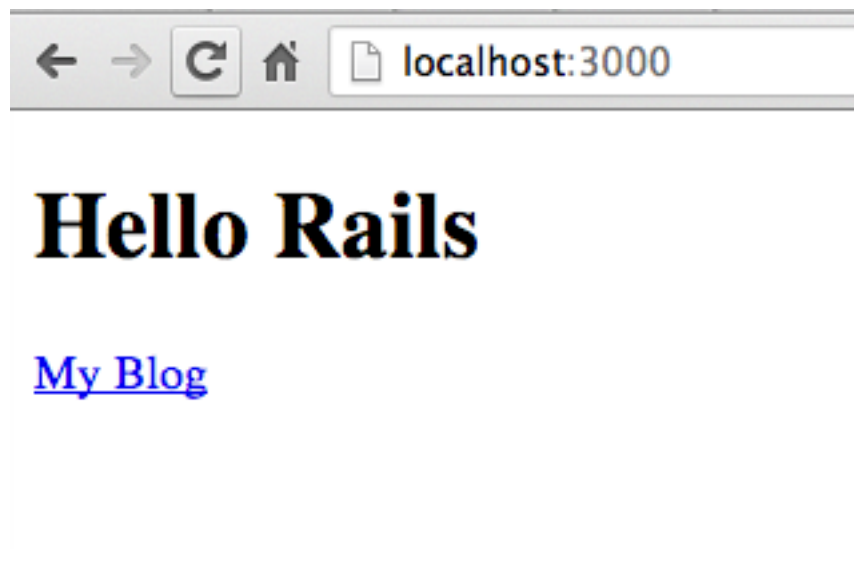


Figure 21: My Blog

You will see the hyper link in the home page.

Step 4

Right click and do 'View Page Source'.



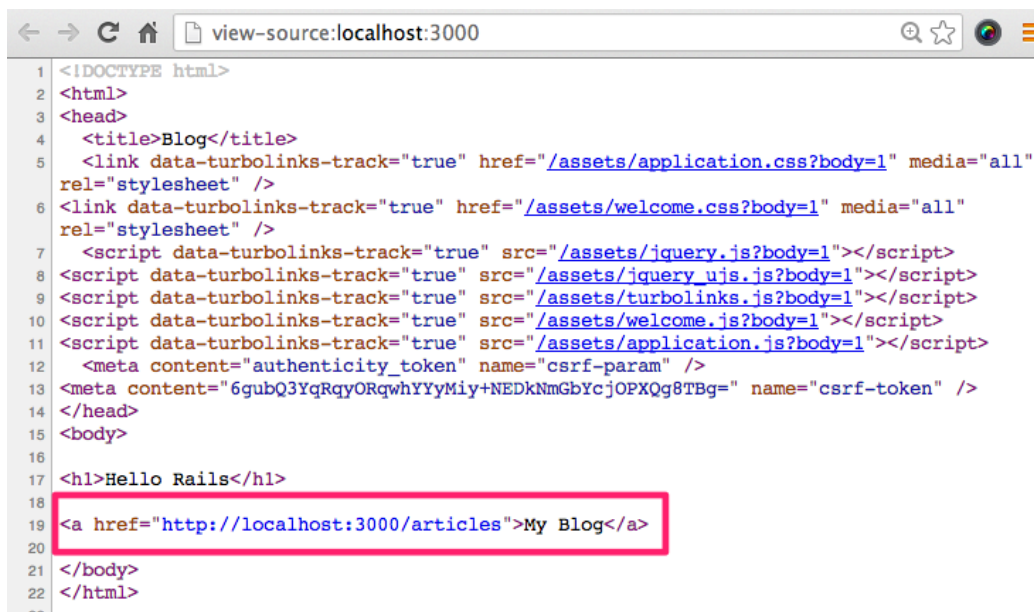
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Blog</title>
5   <link data-turbolinks-track="true" href="/assets/application.css?body=1" media="all"
6   rel="stylesheet" />
7   <link data-turbolinks-track="true" href="/assets/welcome.css?body=1" media="all"
8   rel="stylesheet" />
9   <script data-turbolinks-track="true" src="/assets/jquery.js?body=1"></script>
10  <script data-turbolinks-track="true" src="/assets/jquery_ujs.js?body=1"></script>
11  <script data-turbolinks-track="true" src="/assets/turbolinks.js?body=1"></script>
12  <script data-turbolinks-track="true" src="/assets/welcome.js?body=1"></script>
13  <script data-turbolinks-track="true" src="/assets/application.js?body=1"></script>
14  <meta content="authenticity_token" name="csrf-param" />
15  <meta content="6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=" name="csrf-token" />
16 </head>
17 <body>
18   <h1>Hello Rails</h1>
19   <a href="/articles">My Blog</a>
20 </body>
21 </html>
```

Figure 22: Relative URL

You will see the hyperlink which is a relative url.

Step 5

Change the `articles_path` to `articles_url` in the `welcome/index.html.erb`.



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Blog</title>
5   <link data-turbolinks-track="true" href="/assets/application.css?body=1" media="all"
   rel="stylesheet" />
6   <link data-turbolinks-track="true" href="/assets/welcome.css?body=1" media="all"
   rel="stylesheet" />
7   <script data-turbolinks-track="true" src="/assets/jquery.js?body=1"></script>
8   <script data-turbolinks-track="true" src="/assets/jquery_ujs.js?body=1"></script>
9   <script data-turbolinks-track="true" src="/assets/turbolinks.js?body=1"></script>
10  <script data-turbolinks-track="true" src="/assets/welcome.js?body=1"></script>
11  <script data-turbolinks-track="true" src="/assets/application.js?body=1"></script>
12  <meta content="authenticity_token" name="csrf-param" />
13  <meta content="6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=" name="csrf-token" />
14 </head>
15 <body>
16
17 <h1>Hello Rails</h1>
18
19 <a href="http://localhost:3000/articles">My Blog</a>
20
21 </body>
22 </html>
```

Figure 23: Absolute URL

View page source to see the absolute URL.

Step 6

Click on the 'My Blog' link.

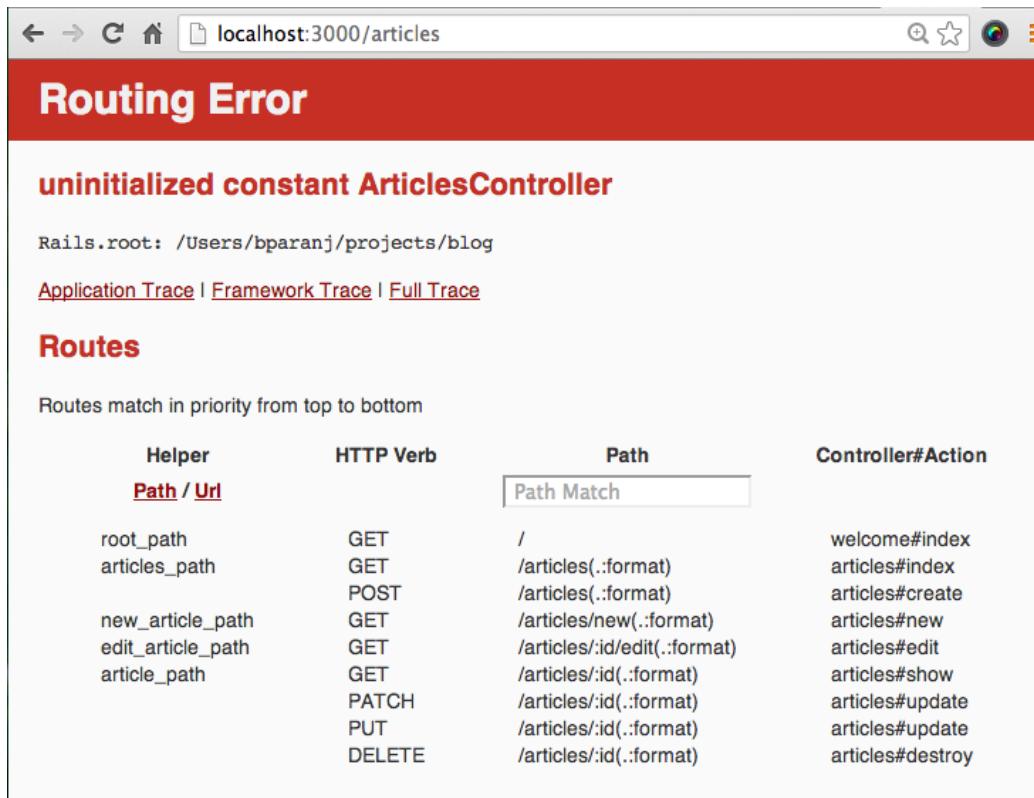
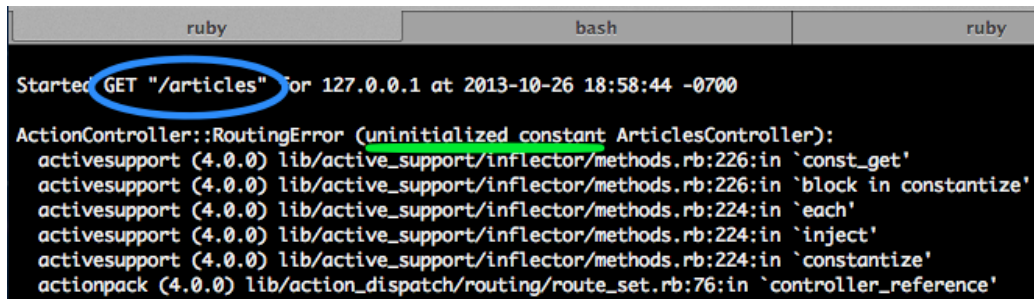


Figure 24: Missing Articles Controller

You will see the above error page.

Step 7

When you click on that link, you can see from rails server log that the client made a request:



```
Started GET "/articles" for 127.0.0.1 at 2013-10-26 18:58:44 -0700
ActionController::RoutingError (uninitialized constant ArticlesController):
  activesupport (4.0.0) lib/active_support/inflector/methods.rb:226:in `const_get'
  activesupport (4.0.0) lib/active_support/inflector/methods.rb:226:in `block in constantize'
  activesupport (4.0.0) lib/active_support/inflector/methods.rb:224:in `each'
  activesupport (4.0.0) lib/active_support/inflector/methods.rb:224:in `inject'
  activesupport (4.0.0) lib/active_support/inflector/methods.rb:224:in `constantize'
  actionpack (4.0.0) lib/action_dispatch/routing/route_set.rb:76:in `controller_reference'
```

Figure 25: Articles Http Request

GET ‘/articles’ that was recognized by the Rails router and it looked for articles controller. Since we don’t have the articles controller, we get the error message for the uninitialized constant. In Ruby, class names are constant.

Live HTTP Headers			
<div>Capture Raw Clear Settings</div>			
#	Method	Status	Url
2	GET	200	http://localhost:3000/favicon.ico
1	GET	200	http://localhost:3000/articles

Figure 26: Live HTTP Headers Client Server Interaction

You can also use HTTP Live Headers Chrome plugin to see the client and server interactions.

```
Headers
GET /articles HTTP/1.1
Host: localhost:3000
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Cookie: request_method=GET; _blog_session=L0dna0EvcGgrUTJabXhucUMvZ3o3MzBzVEZNdWJFZk
Referer: http://localhost:3000/
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.36 (KHTML
HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Content-Length: 1129
Content-Type: text/html; charset=utf-8
Date: Sun, 27 Oct 2013 05:41:56 GMT
Etag: "ff6a5901a468bde7fb289673dc7a7dd6"
Server: WEBrick/1.3.1 (Ruby/2.0.0/2013-06-27)
Set-Cookie: _blog_session=QVFTQ2NiM2gvN0dXRnI0MnpJc2QvbmZXTmFEVmVzcDVCWUVteFRWeDAvRk
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Request-Id: 7f5038d4-790c-413e-9224-10ec973bedfe
X-Runtime: 0.016024
X-Ua-Compatible: chrome=1
X-Xss-Protection: 1; mode=block
```

Figure 27: Live HTTP Headers Showing Client Server Interaction

Here you see the client-server interaction details.

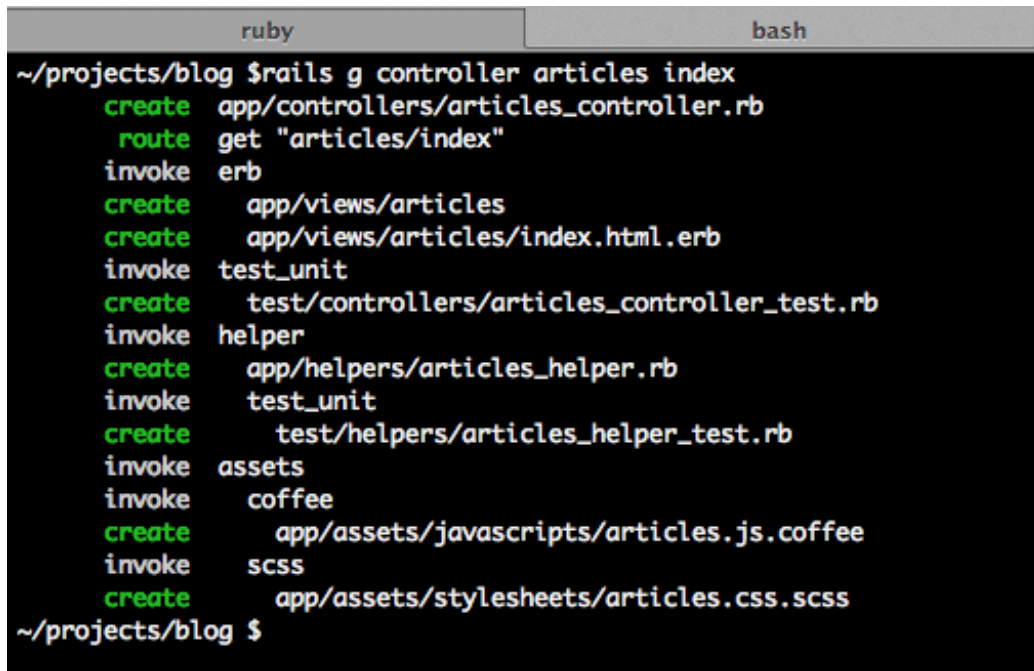
You can learn a lot by looking at the Live HTTP Header details such as Etag which is used for caching by Rails.

Headers	
GET http://localhost:3000/articles Status: HTTP/1.1 200 OK	
Request Headers	
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding	gzip, deflate, sdch
Accept-Language	en-US,en;q=0.8
Cookie	request_method=GET; _blog_session=L0dna0EvoGgrUTJabXhucUMvZ3o3MzBzVEZNdWJFZk9hYIFS-05c21ea3d19f3949a467deb04d54301841302ff1
Referer	http://localhost:3000/
User-Agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.36 (KHTML,
Response Headers	
Cache-Control	max-age=0, private, must-revalidate
Content-Length	1129
Content-Type	text/html; charset=utf-8
Date	Sun, 27 Oct 2013 05:41:56 GMT
Etag	"ff6a5901a468bde7fb289673dc7a7dd6"
Server	WEBrick/1.3.1 (Ruby/2.0.0/2013-06-27)
Set-Cookie	_blog_session=QVFTQ2NiM2gvN0dXRnI0MnpJc2QvbmZXTmFEVmVzcDVCW-4903104c2800dfcd11eeba144af0c6cbc9bb4f53; path=/; HttpOnly
X-Content-Type-Options	nosniff
X-Frame-Options	SAMEORIGIN
X-Request-Id	7f5038d4-790c-413e-9224-10ec973bedfe
X-Runtime	0.016024
X-Ua-Compatible	chrome=1
X-Xss-Protection	1; mode=block

Figure 28: Live HTTP Headers Gives Ton of Information

Step 8

Create the articles controller by running the following command in the blog directory:

A terminal window with a dark background and light-colored text. The window has two tabs at the top: 'ruby' and 'bash'. The prompt is '~ /projects/blog'. The command '\$ rails g controller articles index' has been entered. The output shows the generation of several files and the invocation of various Rails components. The files created are: 'app/controllers/articles_controller.rb', 'app/views/articles/index.html.erb', 'test/controllers/articles_controller_test.rb', 'app/helpers/articles_helper.rb', 'test/helpers/articles_helper_test.rb', and 'app/assets/stylesheets/articles.css.scss'. The components invoked are: 'erb', 'test_unit', 'helper', 'assets', 'coffee', and 'scss'. The terminal ends with the prompt '~ /projects/blog \$'.

```
~/projects/blog $ rails g controller articles index
  create  app/controllers/articles_controller.rb
  route   get "articles/index"
  invoke  erb
  create  app/views/articles
  create  app/views/articles/index.html.erb
  invoke  test_unit
  create  test/controllers/articles_controller_test.rb
  invoke  helper
  create  app/helpers/articles_helper.rb
  invoke  test_unit
  create  test/helpers/articles_helper_test.rb
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/articles.js.coffee
  invoke  scss
  create  app/assets/stylesheets/articles.css.scss
~/projects/blog $
```

Figure 29: Generate Controller

```
$ rails g controller articles index
```


Step 9

Go back to the home page and click on My Blog link.



Figure 30: Articles Page

You will see a static page.

Step 10

We need to replace the static page with the list of articles from the database. Open the `articles_controller.rb` and change the index method as follows :

```
def index
  @articles = Article.all
end
```

Here the `@articles` is an instance variable of the articles controller class. It is made available to the corresponding view class. In this case the view is `app/views/articles/index.html.erb`

The class method `all` retrieves all the records from the articles table.

Step 11

Open the `app/views/articles/index.html.erb` in your IDE and add the following code:

```
<h1>Listing Articles</h1>

<% @articles.each do |article| %>

  <%= article.title %> <br/>

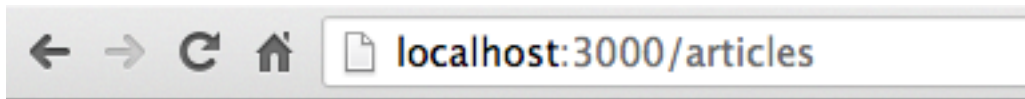
  <%= article.description %>

<% end %>
```

Here we are using the Ruby scriptlet tag `<% %>` for looping through all the records in the articles collection and the values of each record is displayed using `<%= %>` tags.

Step 12

Go to the browser and reload the page for <http://localhost:3000/articles>



Listing Articles

```
test
first row another record
different way to create row
```

Figure 31: List of Articles

You should see the list of articles now displayed in the browser.

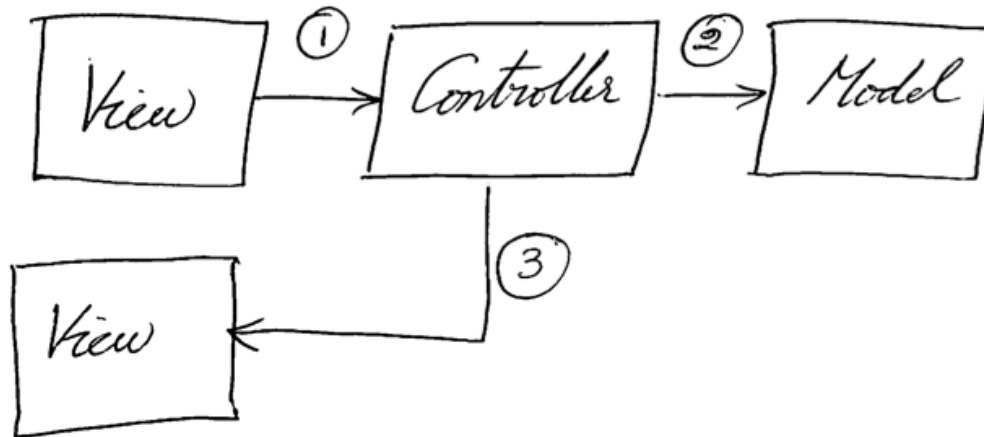


Figure 32: Model View Controller

As you can see from the diagram Controller controls the flow of data into and out of the database and also decides which View should be rendered next.

Summary

In this lesson we went from the view (home page) to the controller for articles and to the article model and back to the view (index page for articles). So the MVC components interaction as shown in the diagram:

1. View to Controller
2. Controller to Model
3. Controller to View.

The data flow was from the database to the user.

In real world the user data comes from the user so we cannot create them in the rails console or in the database directly. In the next lesson we will see how we can capture data from the view provided by the user and save it in the database.

Exercise

Go to the rails server log terminal, what is the http verb used to make the request for displaying all the articles? What is the resource that was requested?

5. View to Model

Objective

- Learn how to get data from the user and save it in the database

Steps

Step 1

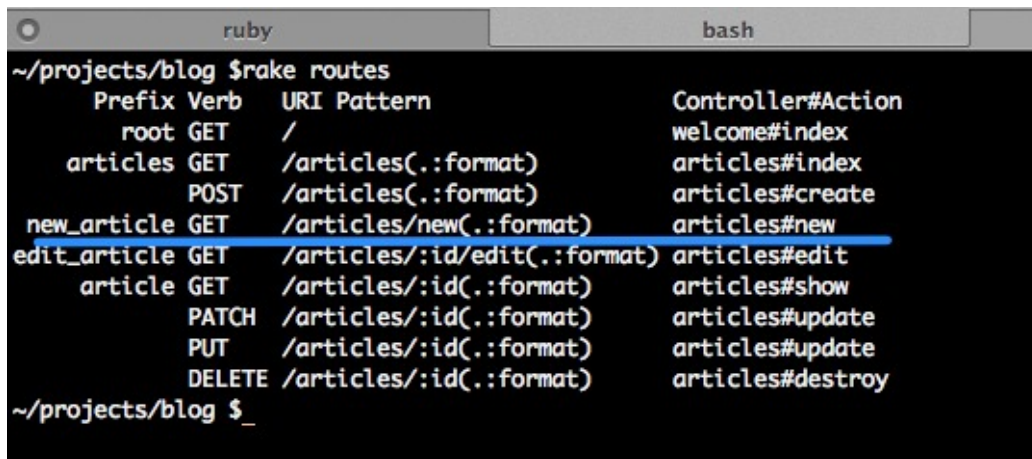
We need to display a form for the user to fill out the text field for the article title and text area for the description. In order for the user to go to this form, let's create a 'New Article' link to load an empty form in the articles index page.

Add the following code to the bottom of the `app/views/articles/index.html` file:

```
<%= link_to 'New Article', ? %>
```

Step 2

What is the url helper we should use? We know we need to display the `articles/new.html.erb` page. We also know that the action that is executed is `new` before `new.html.erb` is displayed. Take a look at the rake routes output:



```
~/projects/blog $rake routes
  Prefix Verb   URI Pattern               Controller#Action
  root   GET    /                       welcome#index
  articles GET    /articles(:format)       articles#index
         POST   /articles(:format)       articles#create
  new_article GET    /articles/new(:format)   articles#new
  edit_article GET    /articles/:id/edit(:format) articles#edit
  article GET    /articles/:id(:format)   articles#show
         PATCH /articles/:id(:format)   articles#update
         PUT    /articles/:id(:format)   articles#update
         DELETE /articles/:id(:format)   articles#destroy
~/projects/blog $
```

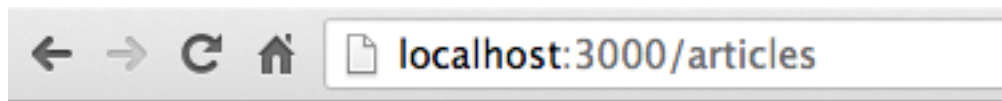
Figure 33: New Article URL Helper

The first column named Prefix gives us the URL helper we can use. We can either append url or path to the prefix. Let's fill in the url helper to load the new page as follows:

```
<%= link_to 'New Article', new_article_path %>
```

Step 3

Reload the page `http://localhost:3000/articles` in the browser.



Listing Articles

test

first row another record

different way to create row

[New Article](#)

Figure 34: New Article Link

The hyperlink for creating a new article will now be displayed.

Step 4

Right click on the browser and click 'View Page Source'.

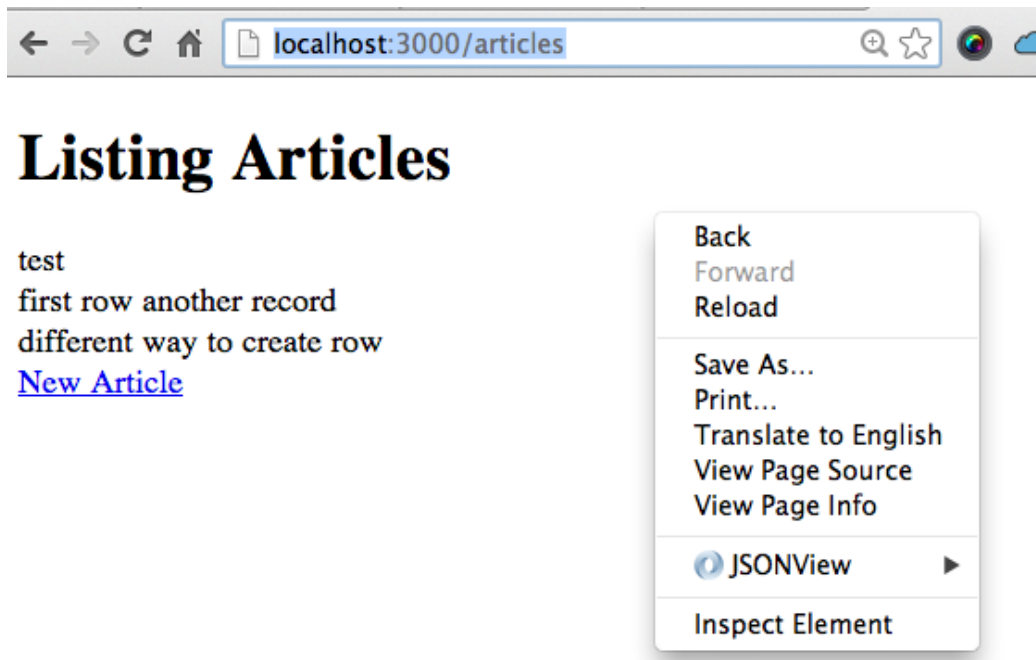
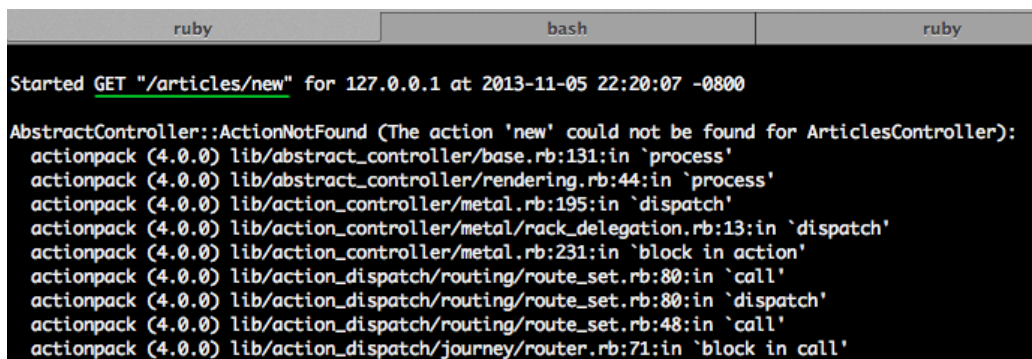


Figure 35: View Page Source

You will see 'New Article' link pointing to the resource “/articles/new”.

Step 5

Click the ‘New Article’ link. Go to the terminal and look at the server output.



```
Started GET "/articles/new" for 127.0.0.1 at 2013-11-05 22:20:07 -0800

AbstractController::ActionNotFound (The action 'new' could not be found for ArticlesController):
  actionpack (4.0.0) lib/abstract_controller/base.rb:131:in `process'
  actionpack (4.0.0) lib/abstract_controller/rendering.rb:44:in `process'
  actionpack (4.0.0) lib/action_controller/metal.rb:195:in `dispatch'
  actionpack (4.0.0) lib/action_controller/metal/rack_delegation.rb:13:in `dispatch'
  actionpack (4.0.0) lib/action_controller/metal.rb:231:in `block in action'
  actionpack (4.0.0) lib/action_dispatch/routing/route_set.rb:80:in `call'
  actionpack (4.0.0) lib/action_dispatch/routing/route_set.rb:80:in `dispatch'
  actionpack (4.0.0) lib/action_dispatch/routing/route_set.rb:48:in `call'
  actionpack (4.0.0) lib/action_dispatch/journey/router.rb:71:in `block in call'
```

Figure 36: HTTP Verb Get

You can see that the browser made a http GET request for the resource “/articles/new”.



Figure 37: Action New Not Found

You will see the above error page.

Step 6

Let's create the new action in articles controller. Add the following code to articles controller:

```
def new  
  
end
```

Step 7

Reload the browser <http://localhost:3000/articles/new> page. You will see the missing template page.

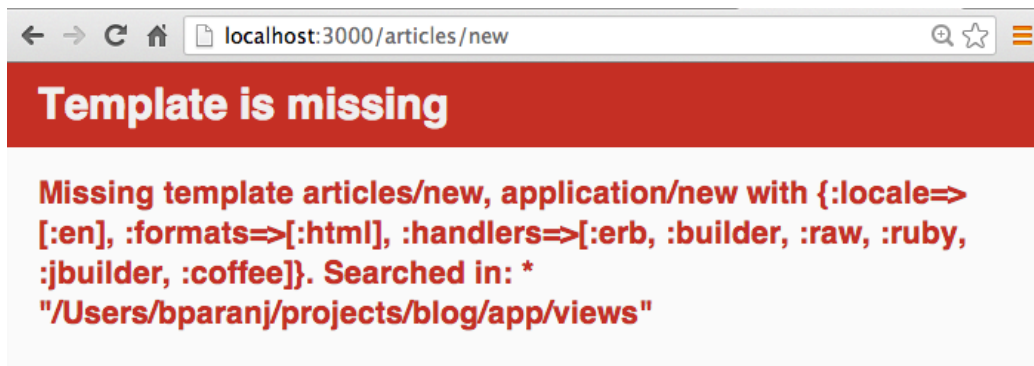


Figure 38: Missing Template

After the new action is executed Rails looks for view whose name is the same as the action, in this case `app/views/articles/new.html.erb`

Step 8

So lets create new.html.erb under app/views/articles directory with the following content:

```
<%= form_for @article do |f| %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :description %><br>
    <%= f.text_area :description %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Step 9

Reload the browser <http://localhost:3000/articles/new> page.

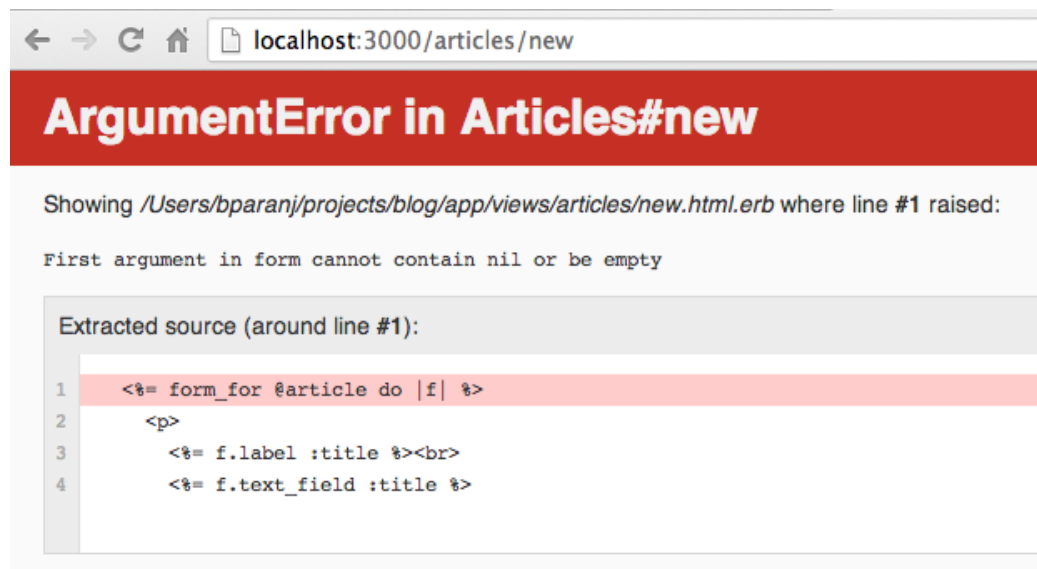


Figure 39: Argument Error

You will now see the above error.

Step 10

Change the new method in articles controller as follows:

```
def new
  @article = Article.new
end
```

Here we are instantiating an instance of Article class, this gives Rails a clue that the form fields is for Article model.

Step 11

Reload the browser `http://localhost:3000/articles/new` page.



The screenshot shows a web browser window with the address bar displaying `localhost:3000/articles/new`. Below the address bar, the page content includes a 'Title' label followed by a single-line text input field. Below that is a 'Description' label followed by a multi-line text input field. At the bottom of the form is a button labeled 'Save Article'.

Figure 40: New Article Form

You will now see an empty form to create a new article.

Step 12

Right click and select 'View Page Source' on the new article form page.



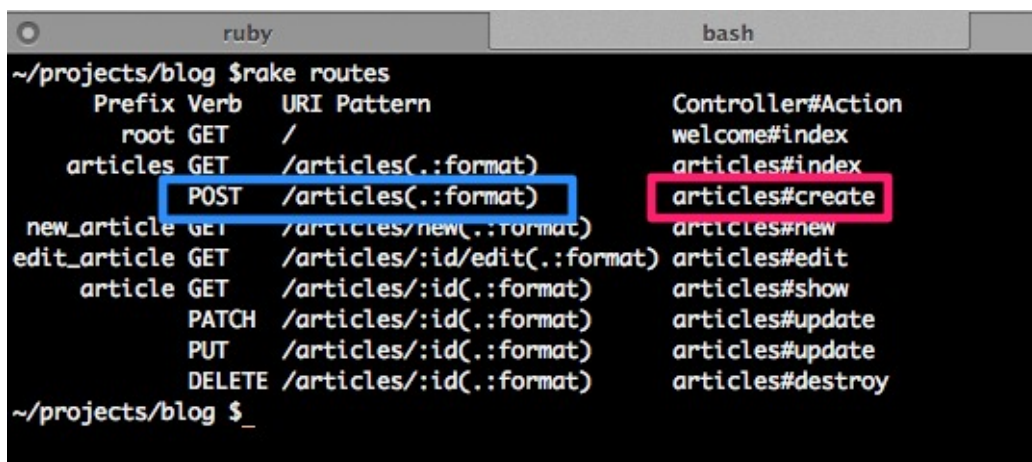
```
17 <body>
18
19 <form accept-charset="UTF-8" action="/articles" class="new_article"
id="new_article" method="post"><div style="margin:0;padding:0;display:inline">
<input name="utf8" type="hidden" value="&#x2713;" /><input
name="authenticity_token" type="hidden"
value="6gubQ3YqRqyORqwhYyMiy+NEDkNmGbYcjOPXQg8TBg=" /></div>
20   <p>
21     <label for="article_title">Title</label><br>
22     <input id="article_title" name="article[title]" type="text" />
23   </p>
24
25   <p>
26     <label for="article_description">Description</label><br>
27     <textarea id="article_description" name="article[description]">
28   </textarea>
29   </p>
30
31   <p>
32     <input name="commit" type="submit" value="Create Article" />
33   </p>
34 </form>
35
36 </body>
```

Figure 41: New Article Page Source

As you can see form will be submitted to the url '/articles' and the http verb used is POST. When the user submits the form, which controller and which action will be executed?

Step 13

Look at the output of rake routes, the combination of the http verb and the URL uniquely identifies the resource end point.



The image shows a terminal window with two tabs: 'ruby' and 'bash'. The 'ruby' tab is active, and the command `$rake routes` has been executed. The output is a table with four columns: Prefix, Verb, URI Pattern, and Controller#Action. The row for the POST verb to the /articles URI is highlighted with a blue box, and the corresponding Controller#Action 'articles#create' is highlighted with a red box.

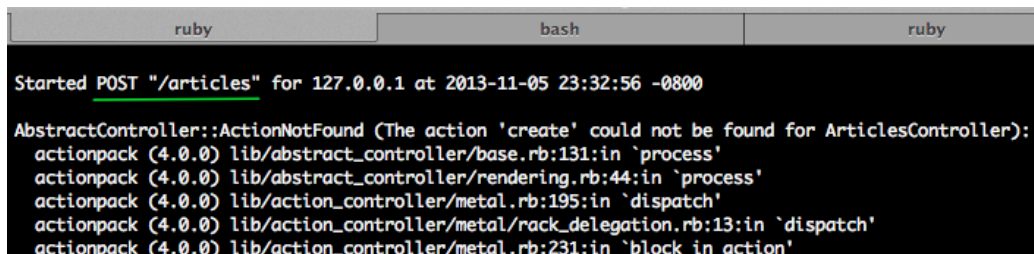
Prefix	Verb	URI Pattern	Controller#Action
root	GET	/	welcome#index
articles	GET	/articles(.:format)	articles#index
	POST	/articles(.:format)	articles#create
new_article	GET	/articles/new(.:format)	articles#new
edit_article	GET	/articles/:id/edit(.:format)	articles#edit
article	GET	/articles/:id(.:format)	articles#show
	PATCH	/articles/:id(.:format)	articles#update
	PUT	/articles/:id(.:format)	articles#update
	DELETE	/articles/:id(.:format)	articles#destroy

Figure 42: Identifying Resource Endpoint

In this case we see that it maps to the articles controller and create action.

Step 14

Fill out the form and click 'Create Article'. Check the server log output.



```
Started POST "/articles" for 127.0.0.1 at 2013-11-05 23:32:56 -0800
AbstractController::ActionNotFound (The action 'create' could not be found for ArticlesController):
  actionpack (4.0.0) lib/abstract_controller/base.rb:131:in `process'
  actionpack (4.0.0) lib/abstract_controller/rendering.rb:44:in `process'
  actionpack (4.0.0) lib/action_controller/metal.rb:195:in `dispatch'
  actionpack (4.0.0) lib/action_controller/metal/rack_delegation.rb:13:in `dispatch'
  actionpack (4.0.0) lib/action_controller/metal.rb:231:in `block in action'
```

Figure 43: Post Article Server Log

You can see that the browser made a post to the URL '/articles'.



Figure 44: Unknown Action Create

This error is due to absence of create action in the articles controller.

Step 15

Define the create method in the articles controller as follows:

```
def create  
  
end
```

Step 16

Fill out the form and click 'Create Article'.

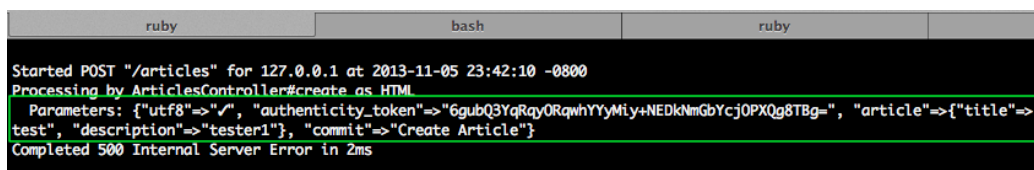


Figure 45: Article Form Values

You can see that the form values submitted by the user is sent to the server. Rails automatically populates a hash called params which contains a key whose name is the article symbol and the values are the different columns and its values.



Figure 46: Article Create Missing Template

You will see missing tempate error.

Step 17

Before we fix the missing template issue, we need to save the data submitted by the user in the database. You already know how to use the ActiveRecord class method create to save a record. You also know how Rails populates the params hash, this hash is made available to you in the controller. So we can access it like this :

```
def create
  Article.create(params[:article])
end
```

Step 18

Fill out the form and submit again.

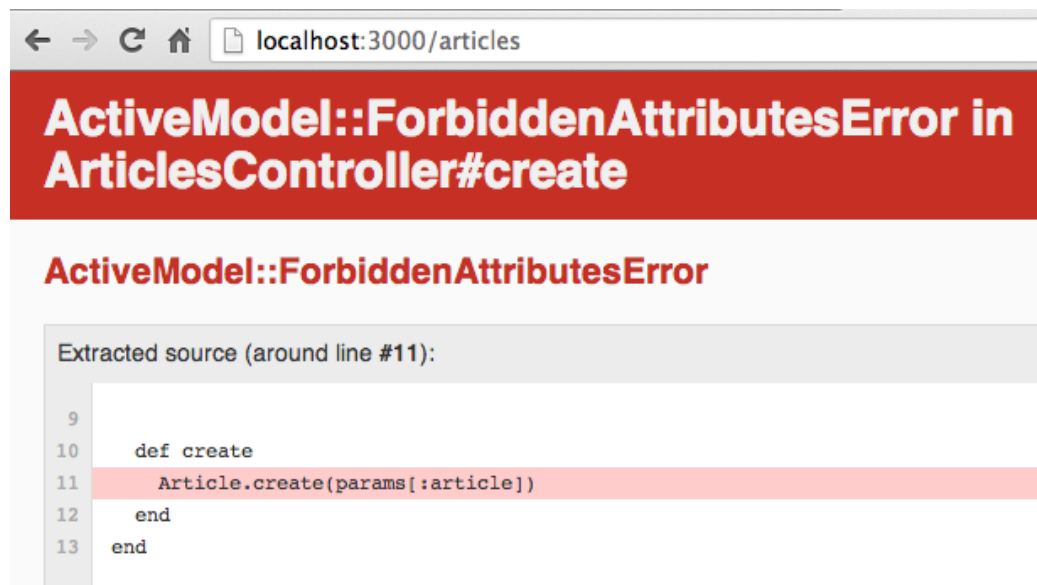


Figure 47: Forbidden Attributes Error

Now we get a forbidden attributes error.

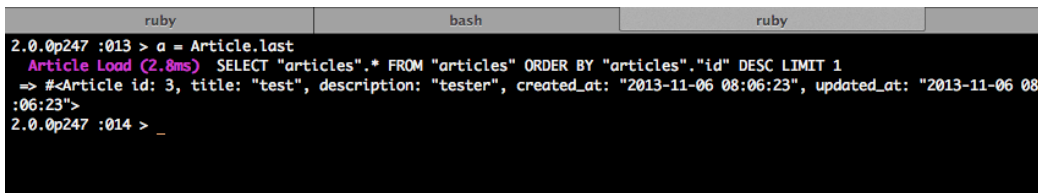
Step 19

Due to security reasons we need to specify which fields must be permitted as part of the form submission. Modify the create method as follows:

```
def create
  Article.create(params.require(:article).permit(:title, :description))
end
```

Step 20

Fill out the form and submit again. You will get the template missing error but you will now see that the user submitted data has been saved to the database.

A terminal window with a dark background and light-colored text. It shows a Ruby IRB session. The prompt is '2.0.0p247 :013 >'. The user enters 'a = Article.last'. The prompt changes to 'Article Load (2.8ms)'. The user enters 'SELECT "articles".* FROM "articles" ORDER BY "articles"."id" DESC LIMIT 1'. The prompt changes to '=> #<Article id: 3, title: "test", description: "tester", created_at: "2013-11-06 08:06:23", updated_at: "2013-11-06 08:06:23">'. The user enters ':014 > _'. The prompt changes to '2.0.0p247 :014 > _'.

```
2.0.0p247 :013 > a = Article.last
Article Load (2.8ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id" DESC LIMIT 1
=> #<Article id: 3, title: "test", description: "tester", created_at: "2013-11-06 08:06:23", updated_at: "2013-11-06 08:06:23">
2.0.0p247 :014 > _
```

Figure 48: Save User Data

The ActiveRecord last method retrieves the last row in the articles table.

Step 21

Let's now address the template is missing error. Once the data is saved in the database we can either display the index page or the show page for the article. Let's redirect the user to the index page. We will be able to see all the records including the new record that we created. Modify the create method as follows:

```
def create
  Article.create(params.require(:article).permit(:title, :description))

  redirect_to articles_index_path
end
```

How do we know that we need to use `articles_index_path` url helper? We already saw how to find the URL helper to use in the view, we can do the same. If you see the output of `rake routes` command, we know the resource end point, to find the URL helper we look at the Prefix column.

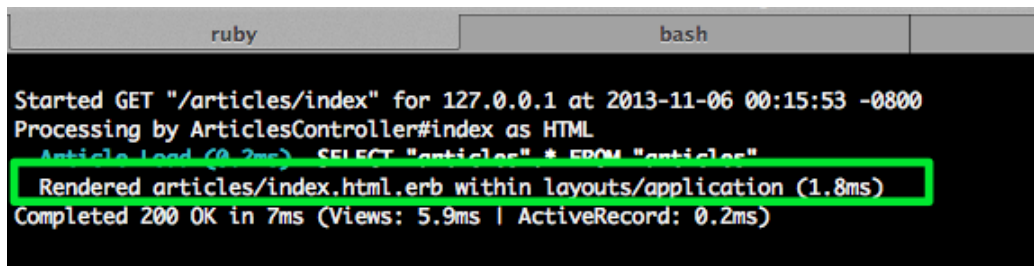
Step 22

Fill out the form and submit again.



Figure 49: Displaying All Articles

You will now see all the articles displayed in the index page.



```
ruby bash
Started GET "/articles/index" for 127.0.0.1 at 2013-11-06 00:15:53 -0800
Processing by ArticlesController#index as HTML
Article Load (0.2ms) SELECT "articles" * FROM "articles"
Rendered articles/index.html.erb within layouts/application (1.8ms)
Completed 200 OK in 7ms (Views: 5.9ms | ActiveRecord: 0.2ms)
```

Figure 50: Redirect to Articles Index Page

Redirecting user to the articles index page.

Summary

We saw how we can save the user submitted data in the database. We went from the View to the Controller to the Model. We also saw how the controller decides which view to render next. We learned about the http verb and identifying resource endpoint in our application. Now we know how the new and create works. In the next lesson we will see how edit and update works to make changes to an existing record in the database.

6. Update Article

Objective

- Learn how to update an existing record in the database

Steps

Step 1

Let's add 'Edit' link to each record that is displayed in the index page. Open the app/views/index.html.erb and add the edit link:

```
<%= link_to 'Edit', ? %>
```

What should be the url helper to use in the second parameter to the link_to method? We know that when someone clicks the 'Edit' link we need to load a form for that particular row with the existing values for that record. So we know the resource endpoint is articles#edit, if you look at the rake routes output, the Prefix column gives us the url helper to use.

ruby				bash	
~/projects/blog \$rake routes					
	Prefix	Verb	URI Pattern	Controller#Action	
	root	GET	/	welcome#index	
	articles	GET	/articles(.:format)	articles#index	
		POST	/articles(.:format)	articles#create	
	new_article	GET	/articles/new(.:format)	articles#new	
	edit_article	GET	/articles/:id/edit(.:format)	articles#edit	
	article	GET	/articles/:id(.:format)	articles#show	
		PATCH	/articles/:id(.:format)	articles#update	
		PUT	/articles/:id(.:format)	articles#update	
		DELETE	/articles/:id(.:format)	articles#destroy	
~/projects/blog \$_					

Figure 51: Edit Article URL Helper

So we now have:

```
<%= link_to 'Edit', edit_article_path() %>
```

In the URI Pattern column you see the pattern for edit as : /articles/:id/edit. URI Pattern can consist of symbols which represent variable. You can think of it as a place holder. The symbol :id in this case represents the primary key of the record we want to update. So we pass an instance of article to url helper. We could call the id method on article, since Rails automatically calls id on this instance, we will just let Rails do its magic.

```
<%= link_to 'Edit', edit_article_path(article) %>
```

The app/views/articles/index.html.erb will look like this :

```
<h1>Listing Articles</h1>
```

```
<% @articles.each do |article| %>
```

```
  <%= article.title %> :
```

```
  <%= article.description %>
```

```
  <%= link_to 'Edit', edit_article_path(article) %>
```

```
  <br/>
```

```
<% end %>
```

```
<br/>
```

```
<%= link_to 'New Article', new_article_path %>
```


Step 2

Reload the `http://localhost:3000/articles` page.



Figure 52: Edit Article Link

You will now see the ‘Edit’ link for each article in the database.

Step 3

Right click on the browser and select 'View Page Source'.



```
18
19 <h1>Listing Articles</h1>
20
21
22     test :
23
24     first row
25
26     <a href="/articles/1/edit">Edit</a>
27
28     <br/>
29
30
31     another record :
32
33     different way to create row
34
35     <a href="/articles/2/edit">Edit</a>
36
37     <br/>
38
```

Figure 53: Edit Article Page Source

You will see the primary keys of the corresponding row for the :id variable.

Step 4

Click on the 'Edit' link.



Figure 54: Unknown Action Edit

You will see unknown action edit error page.

Step 5

Let's define the edit action in the articles controller :

```
def edit  
  
end
```

Step 6

Click on the 'Edit' link. You now get template is missing error. Let's create app/views/articles/edit.html.erb with the following contents:

```
<%= form_for @article do |f| %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :description %><br>
    <%= f.text_area :description %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Step 7

Click on the 'Edit' link. You now get the following error page:



Figure 55: Argument Error in Articles Edit

We have already seen similar error when we implemented the create action.

Step 8

Look at the server log:

```
Started GET "/articles/1/edit" for 127.0.0.1 at 2013-11-06 20:13:41 -0800
Processing by ArticlesController#edit as HTML
Parameters: {"id"=>"1"}
Rendered articles/edit.html.erb within layouts/application (1.4ms)
Completed 500 Internal Server Error in 4ms
```

Figure 56: Edit Article Server Log

You can see that the primary key of the selected article id and it's value.

```
Started GET "/articles/1/edit" for 127.0.0.1 at 2013-11-06 20:13:41 -0800
Processing by ArticlesController#edit as HTML
Parameters: {"id"=>"1"}
Rendered articles/edit.html.erb within layouts/application (1.4ms)
Completed 500 Internal Server Error in 4ms
```

params = { id: 1 }

Figure 57: Params Hash Populated by Rails

Rails automatically populates params hash and makes it available to the controllers.

Step 9

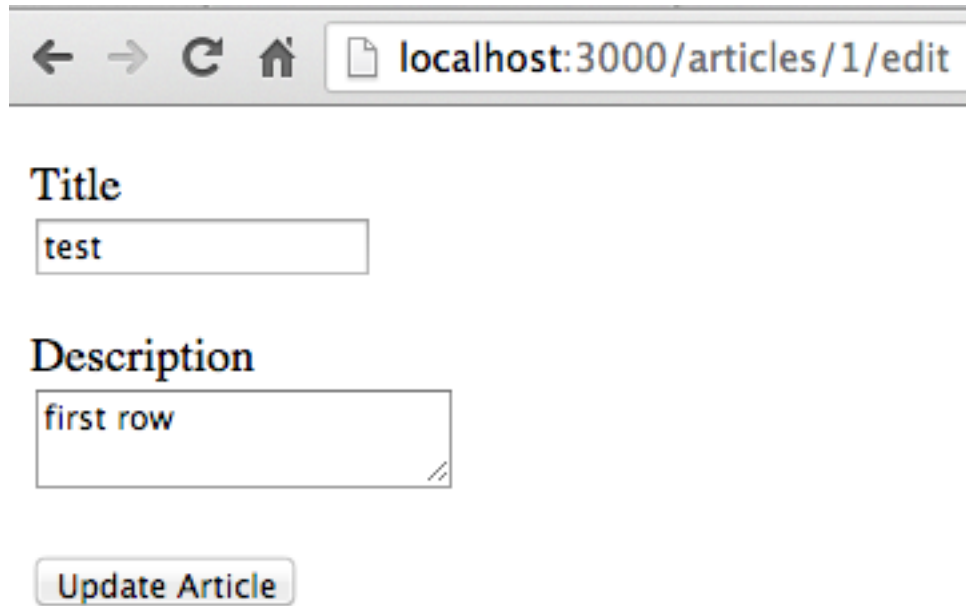
In the edit action we need to load the selected record from the database so that we can display it with the existing values for its columns. You already know that Rails populates params hash with the values submitted in the GET request for resource `‘/articles/1/edit’`. We can now define the edit method as follows:

```
def edit
  @article = Article.find(params[:id])
end
```

Here we find the record for the given primary key and save it in the instance variable `@article`. Since this variable is available in the view, we can now display the record with its existing values.

Step 10

Click on the 'Edit' link.



The screenshot shows a web browser window with the address bar displaying `localhost:3000/articles/1/edit`. Below the address bar, the form is titled "Title" and "Description". The "Title" field contains the text "test". The "Description" field contains the text "first row". At the bottom of the form is a button labeled "Update Article".

Figure 58: Edit Article Form

You will now see the form with values populated.

Step 11

Right click on the browser and click ‘View Page Source’.

```
19 <form accept-charset="UTF-8" action="/articles/1" class="edit_article" id="edit_article_1"
method="post"><div style="margin:0;padding:0;display:inline"><input name="utf8" type="hidden"
value="&#x2713;" /><input name="method" type="hidden" value="patch" /><input
name="authenticity_token" type="hidden" value="6gubQ3YqRqyORqwhYYyMly+NEDkNmGbYcjOPXQg8TBg=" />
</div>
20 <p>
21 <label for="article_title">Title</label><br>
22 <input id="article_title" name="article[title]" type="text" value="test" />
23 </p>
24 <p>
25 <label for="article_description">Description</label><br>
26 <textarea id="article_description" name="article[description]">
27 first row</textarea>
28 </p>
29 <p>
30 <input name="commit" type="submit" value="Update Article" />
31 </p>
32 </form>
```

Figure 59: Edit Article Source

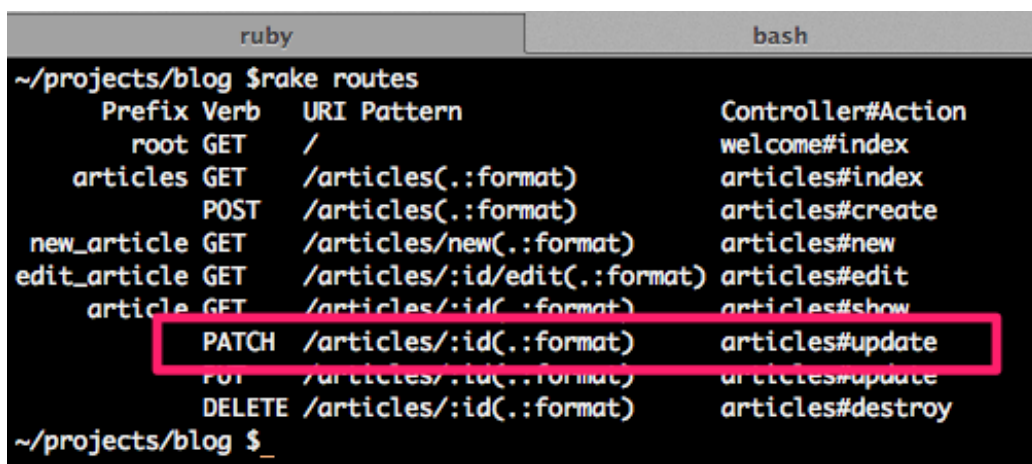
We see that the URI pattern is ‘/articles/1’ and the http verb is POST. If you look at the output of rake routes you will see that POST is used only once for create. The browser knows only GET and POST, it does not know how to use any other http verbs.

ruby			bash
~/projects/blog \$rake routes			
Prefix	Verb	URI Pattern	Controller#Action
root	GET	/	welcome#index
articles	GET	/articles(.:format)	articles#index
	POST	/articles(.:format)	articles#create
new_article	GET	/articles/new(.:format)	articles#new
edit_article	GET	/articles/:id/edit(.:format)	articles#edit
article	GET	/articles/:id(.:format)	articles#show
	PATCH	/articles/:id(.:format)	articles#update
	PUT	/articles/:id(.:format)	articles#update
	DELETE	/articles/:id(.:format)	articles#destroy
~/projects/blog \$ _			

Figure 60: HTTP Verb POST

The question is how to overcome the inability of browsers to speak the entire RESTful vocabulary of using the appropriate http verb for a given operation?

The answer lies in the hidden field called method that has the value PATCH. Rails piggybacks on the POST http verb to actually sneak in a hidden variable that tells the server it is actually a PATCH http verb. If you look at the output of rake routes, for the combination of PATCH and ‘/articles/1’ you will see that it maps to update action in the articles controller.



```
~/projects/blog $rake routes
      Prefix Verb   URI Pattern                      Controller#Action
    root GET    /                               welcome#index
  articles GET    /articles(.:format)             articles#index
          POST   /articles(.:format)             articles#create
 new_article GET    /articles/new(.:format)         articles#new
edit_article GET    /articles/:id/edit(.:format)    articles#edit
 article GET    /articles/:id(.:format)         articles#show
          PATCH  /articles/:id(.:format)         articles#update
          PUT    /articles/:id(.:format)         articles#update
          DELETE /articles/:id(.:format)         articles#destroy
~/projects/blog $
```

Figure 61: HTTP Verb PATCH

The output of rake routes is your friend.

Step 12

Let's implement the update method that will take the new values provided by user for the existing record and update it in the database.

```
def update
  @article = Article.find(params[:id])
  @article.update_attributes(params[:article])
end
```

Before we update the record we need to load the existing record from the database. Why? Because the instance variable in the controller will only exist

for one request-response cycle. Since http is stateless we need to retrieve it again before we can update it.

Step 13

Go to articles index page. Click on the 'Edit' link. In the edit form, you can change the value of either the title or description and click 'Update Article'.

Step 14

To fix the forbidden attributes error, we can do the same thing we did for create action. Change the update method as follows:

```
def update
  @article = Article.find(params[:id])
  permitted_columns = params.require(:article).permit(:title, :description)
  @article.update_attributes(permitted_columns)
end
```

Change the title and click 'Update Article'. We see the template is missing but the record has been successfully updated.



```
2.0.0p247 :014 > a = Article.first
Article Load (0.8ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id" ASC LIMIT 1
=> #<Article id: 1, title: "test", description: "first row updated", created_at: "2013-10-27 01:17:59", updated_at: "2013-11-07 04:58:39">
2.0.0p247 :015 >
```

Figure 62: First Article

The ActiveRecord class method first retrieves the first record in the table. In this case we got the first row in the articles table.

Step 15

Let's address the template is missing error. We don't need `update.html.erb`, we can redirect the user to the index page where all the records are displayed. Change the update method as follows:

```
def update
  @article = Article.find(params[:id])
  permitted_columns = params.require(:article).permit(:title, :description)
  @article.update_attributes(permitted_columns)

  redirect_to articles_path
end
```

Step 16

Edit the article and click 'Update Article'. You should see that it now updates the article.

Step 17

An annoying thing about Rails 4 is that when you run the rails generator to create a controller with a given action it also creates an entry in the `routes.rb` which is not required for a RESTful route. Let's delete the following line:

```
get "articles/index"
```

in the `config/routes.rb` file. Update the create method to use the `articles_path` as follows:

```
def create
  Article.create(params.require(:article).permit(:title, :description))

  redirect_to articles_path
end
```

Summary

In this lesson you learned how to update an existing record. In the next lesson we will see how to display a given article.

7. Show Article

Objective

- Learn how to display a selected article in the article show page.

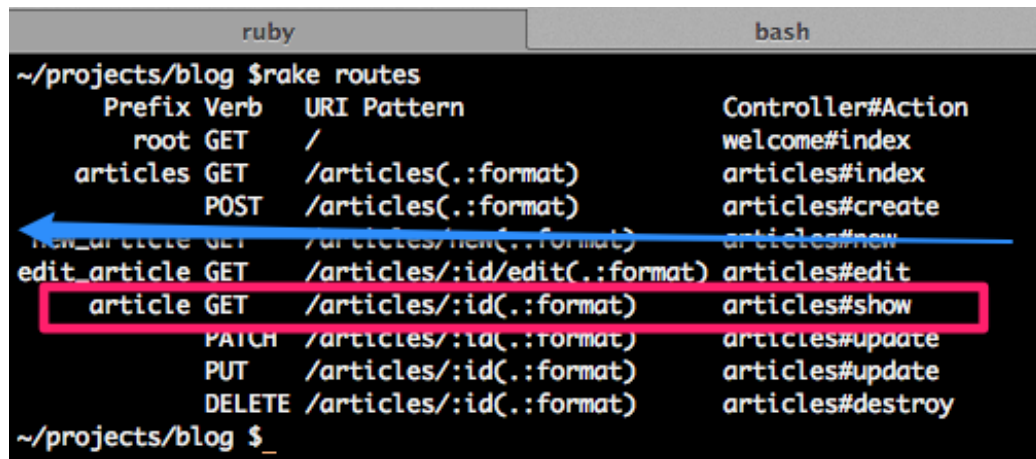
Steps

Step 1

Add the 'Show' link to each article in the index page. The hyperlink text will be 'Show'.

```
<%= link_to 'Show', ? %>
```

When the user clicks the 'Show' link we need to go the articles controller show action. We will retrieve the record from the database and display it in the view. What should be the url helper? You can view the output of rake routes to find the url helper to use in the view. In this case we know the resource end point. We go from the right most column to the left most column and find the url helper under the Prefix column.



```
~/projects/blog $rake routes
      Prefix Verb   URI Pattern               Controller#Action
    root GET    /                  welcome#index
  articles GET    /articles(:format)  articles#index
    articles POST   /articles(:format)  articles#create
new_article GET    /articles/new(:format) articles#new
edit_article GET    /articles/:id/edit(:format) articles#edit
  article GET    /articles/:id(:format) articles#show
    articles PATCH  /articles/:id(:format) articles#update
    articles PUT    /articles/:id(:format) articles#update
    articles DELETE /articles/:id(:format) articles#destroy
~/projects/blog $
```

Figure 63: URL Helper For Show

```
<%= link_to 'Show', article_path %>
```

Since we need the primary key to load the selected article from the database, we need to pass the id as the parameter as below:

```
<%= link_to 'Show', article_path(article.id) %>
```

Since Rails automatically calls the id method of the ActiveRecord we can simplify it as follows:

```
<%= link_to 'Show', article_path(article) %>
```

Rails 4 has optimized this even further so you can do :

```
<%= link_to 'Show', article %>
```

The app/views/articles/index.html.erb looks as shown below:

```
<h1>Listing Articles</h1>
```

```
<% @articles.each do |article| %>
```

```
  <%= article.title %> :
```

```
  <%= article.description %>
```

```
  <%= link_to 'Edit', edit_article_path(article) %>
```

```
  <%= link_to 'Show', article_path(article) %>
```

```
  <br/>
```

```
<% end %>
```

```
<br/>
```

```
<%= link_to 'New Article', new_article_path %>
```

Step 2

Reload the articles index page <http://localhost:3000/articles>



Figure 64: Show Link

You will see the show link.

Step 3

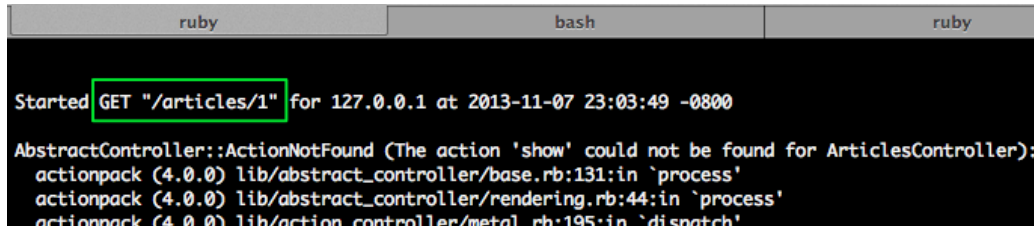
If you view the page source for articles index page, you will see the hyperlink for 'Show' with the URI pattern '/articles/1'. Since this is a hyperlink the browser will use the http verb GET when the user clicks on show.



```
19 <h1>Listing Articles</h1>
20
21
22     test :
23
24     first row updated 2
25
26     <a href="/articles/1/edit">Edit</a>
27     <a href="/articles/1">Show</a>
28
29     <br/>
```

Figure 65: Show Link Source

In the rails server log you will see the GET request for the resource `‘/articles/1’`. In this case the value of `:id` is 1. Rails will automatically populate the params hash with `:id` as the key and the value as the primary key of the record which in this case is 1. We can retrieve the value of the primary key from the params hash and load the record from the database.



```
Started GET "/articles/1" for 127.0.0.1 at 2013-11-07 23:03:49 -0800
AbstractController::ActionNotFound (The action 'show' could not be found for ArticlesController):
  actionpack (4.0.0) lib/abstract_controller/base.rb:131:in `process'
  actionpack (4.0.0) lib/abstract_controller/rendering.rb:44:in `process'
  actionpack (4.0.0) lib/action_controller/metal.rb:195:in `dispatch'
```

Figure 66: Http GET Request

Server log is another friend.

Step 4

If you click on the ‘Show’ link you will get the ‘Unknown action’ error.



Figure 67: Unknown Action Show

As we saw in the previous step, we can get the primary key from the params hash. So, define the show action in the articles controller as follows:

```
def show
  @article = Article.find(params[:id])
end
```

We already know the instance variable `@article` will be made available in the view.

Step 5

If you click the ‘Show’ link, you will get the ‘Template is missing’ error.



Figure 68: Template Missing Error

We need a view to display the selected article. Let’s define `app/views/show.html.erb` with the following content:

```
<p>
  <%= @article.title %><br>
</p>

<p>
  <%= @article.description %><br>
</p>
```

Since the `@article` variable was initialized in the show action, we can retrieve the values of the columns for this particular record and display it in the view. Now the ‘Show’ link will work.

Summary

In this lesson we saw how to display a selected article in the show page. In the next lesson we will see how to delete a given record from the database.

8. Delete Article

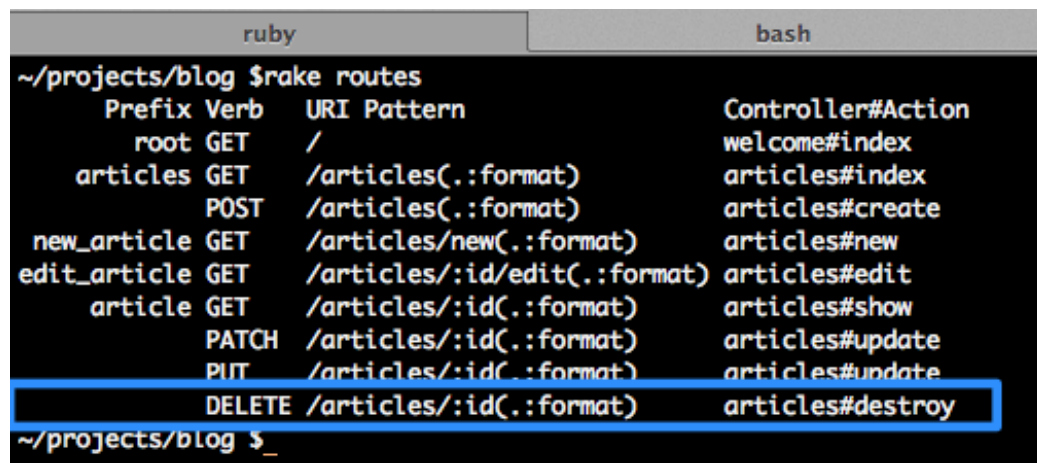
Objectives

- Learn how to delete a given article.
- Learn how to use flash messages.

Steps

Step 1

Let's add 'Delete' link to each record displayed in the articles index page. Look at the output of rake routes.



```
~/projects/blog $rake routes
      Prefix Verb   URI Pattern               Controller#Action
    root GET    /                       welcome#index
  articles GET    /articles(.:format)      articles#index
             POST   /articles(.:format)      articles#create
 new_article GET    /articles/new(.:format)  articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
  article GET    /articles/:id(.:format)  articles#show
             PATCH  /articles/:id(.:format)  articles#update
             PUT    /articles/:id(.:format)  articles#update
             DELETE /articles/:id(.:format)  articles#destroy
~/projects/blog $
```

Figure 69: URL Helper For Delete

The last row is the route for destroy. The Prefix column is empty in this case. It means whatever is above that column that is not empty carries over to that row. So we can create our hyperlink as:

```
<%= link_to 'Delete', article_path(article) %>
```

This will create an hyperlink, when a user clicks on the link the browser will make a http GET request, which means it will end up in show action instead of destroy. Look the Verb column, you see we need to use DELETE http verb to hit the destroy action in the articles controller. So now we have:

```
<%= link_to 'Delete', article_path(article), method: :delete %>
```

The third parameter specifies that the http verb to be used is DELETE. Since this is an destructive action we want to avoid accidental deletion of records, so let's popup a javascript confirmation for delete like this:

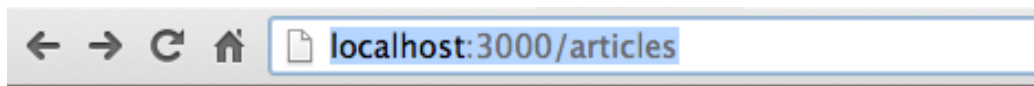
```
<%= link_to 'Delete',  
          article_path(article),  
          method: :delete,  
          data: { confirm: 'Are you sure?' } %>
```

The fourth parameter will popup a window that confirms the delete action. The app/views/articles/index.html.erb now looks like this:

```
<h1>Listing Articles</h1>  
  
<% @articles.each do |article| %>  
  
  <%= article.title %> :  
  
  <%= article.description %>  
  
  <%= link_to 'Edit', edit_article_path(article) %>  
  <%= link_to 'Show', article %>  
  <%= link_to 'Delete',  
            article_path(article),  
            method: :delete,  
            data: { confirm: 'Are you sure?' } %>  
  
  <br/>  
  
<% end %>  
<br/>  
<%= link_to 'New Article', new_article_path %>
```

Step 2

Reload the articles index page <http://localhost:3000/articles>



Listing Articles

test : first row updated 2 [Edit](#) [Show](#) [Delete](#)

another record : different way to create row [Edit](#) [Show](#) [Delete](#)

test : tester [Edit](#) [Show](#) [Delete](#)

testing : again. [Edit](#) [Show](#) [Delete](#)

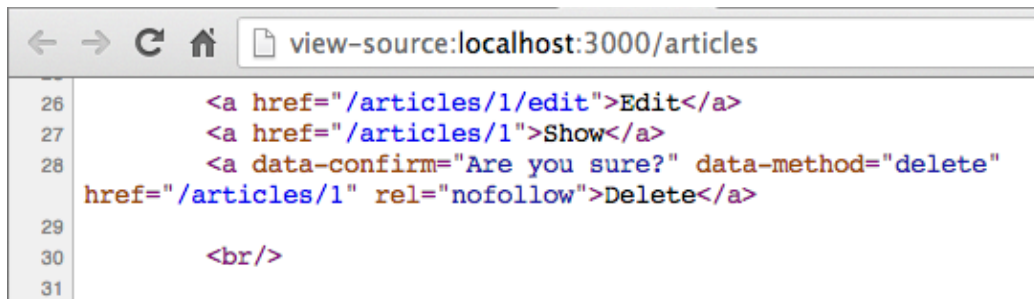
[New Article](#)

Figure 70: Delete Link

The delete link in the browser.

Step 3

In the articles index page, do a ‘View Page Source’.

A screenshot of a web browser's 'View Page Source' window. The address bar shows 'view-source:localhost:3000/articles'. The main content area displays HTML code with line numbers 26 through 31 on the left. The code includes three anchor tags: an 'Edit' link, a 'Show' link, and a 'Delete' link. The 'Delete' link has attributes for a confirmation message, an HTTP method, and a nofollow relationship.

```
26      <a href="/articles/1/edit">Edit</a>
27      <a href="/articles/1">Show</a>
28      <a data-confirm="Are you sure?" data-method="delete"
href="/articles/1" rel="nofollow">Delete</a>
29
30      <br/>
31
```

Figure 71: Delete Link Page Source

You see the html 5 data attribute data-confirm with the value ‘Are you sure?’. This is the text displayed in the confirmation popup window. The data-method attribute value is delete. This is the http verb to be used for this link. The rel=nofollow tells spiders not to crawl these links because it will delete records in the database.

The combination of the URI pattern and the http verb DELETE uniquely identifies a resource endpoint on the server.

Step 4

Right click on the `http://localhost:3000/articles` page. Click on the `jquery_ujs.js` link.

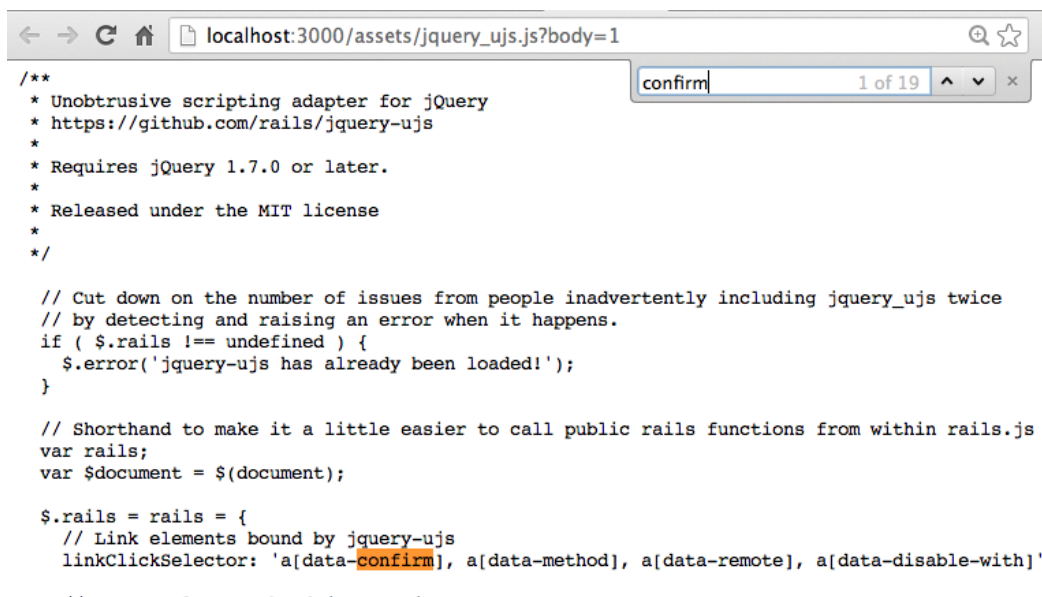


Figure 72: Data Confirm Link Element

Search for ‘confirm’. The first occurrence shows you the link element bound by jquery-ujs. UJS stands for Unobtrusive Javascript. It is unobtrusive because you don’t see any javascript code in the html page.

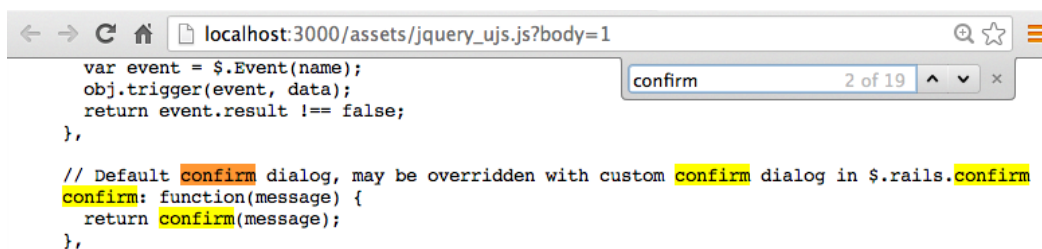


Figure 73: Data Confirm Popup

The second occurrence of the ‘confirm’ shows you the default confirm dialog.

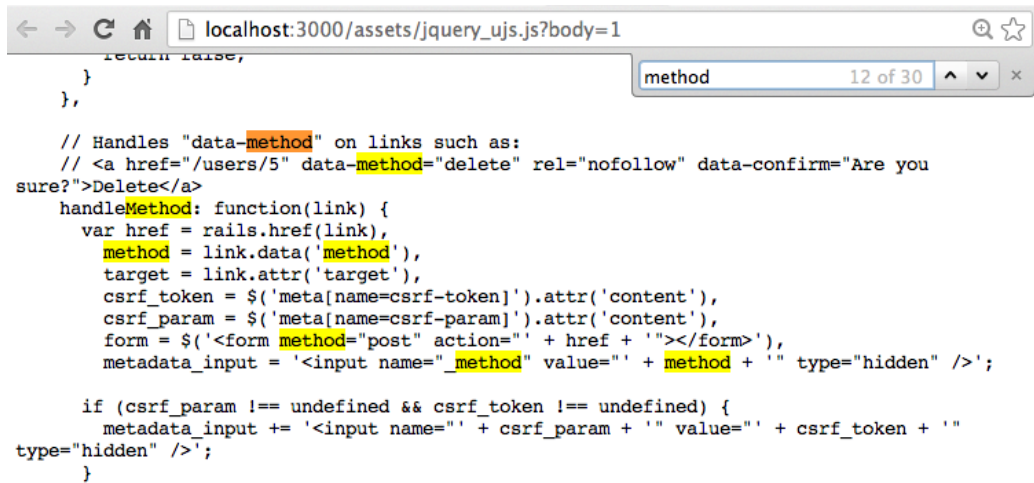


Figure 74: Data Method Delete

You can search for ‘method’. You can see handler method that handles ‘data-method’ on links.

Step 5

In the articles index page, click on the 'Delete' link.

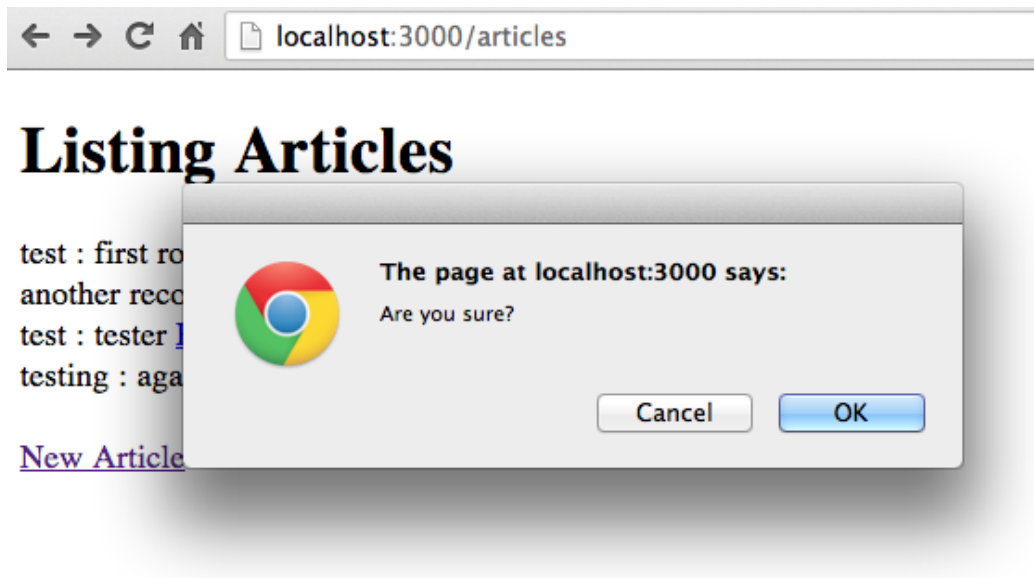


Figure 75: Confirmation Popup

Click 'Cancel'.

Step 6

Define the destroy method in articles controller as follows:

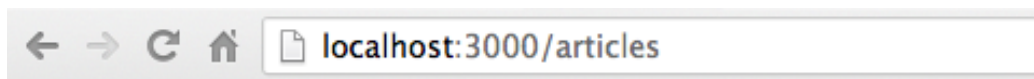
```
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  redirect_to articles_path
end
```

This method is very similar to update method. Instead of updating the record we are deleting it. You already know by this time how to look at the values sent by the browser to the server by looking at the server log output. You also know that params hash will contain the data sent to the server and Rails automatically populates the params hash.

Step 7

In the articles index page, click on the ‘Delete’ link. Click ‘Ok’ in the confirmation popup. The record will now be deleted from the database and you will be redirected back to the articles index page.



Listing Articles

another record : different way to create row [Edit](#) [Show](#) [Delete](#)

test : tester [Edit](#) [Show](#) [Delete](#)

testing : again. [Edit](#) [Show](#) [Delete](#)

[New Article](#)

Figure 76: First Record Deleted

Did we really delete the record?

Step 8

The record was deleted but there is no feedback to the user. Let's modify the destroy action as follows:

```
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  redirect_to articles_path, notice: "Delete success"
end
```

Add the following code after the body tag in the application layout file, app/views/layouts/application.html.erb

```
<% flash.each do |name, msg| -%>
  <%= content_tag :div, msg, class: name %>
<% end -%>
```

Your updated layout file will now look like this:

```
<!DOCTYPE html>
<html>
<head>
<title>Blog</title>
<%= stylesheet_link_tag "application",
media: "all",
"data-turbolinks-track" => true %>
<%= javascript_include_tag "application",
"data-turbolinks-track" => true %>
<%= csrf_meta_tags %>
</head>
<body>

  <% flash.each do |name, msg| -%>
    <%= content_tag :div, msg, class: name %>
  <% end -%>

<%= yield %>

</body>
</html>
```

Step 9

In the articles index page, click on the 'Delete' link.



Figure 77: Delete Success

Now you see the feedback that is displayed to the user after delete operation.

Step 10

In the articles index page, do a ‘View Page Source’.



```
18
19         <div class="notice">Delete success</div>
20
21 <h1>Listing Articles</h1>
22
23
24     test :
25
26     tester
27
28     <a href="/articles/3/edit">Edit</a>
29     <a href="/articles/3">Show</a>
30     <a data-confirm="Are you sure?" data-method="delete"
href="/articles/3" rel="nofollow">Delete</a>
31
32     <br/>
33
```

Figure 78: Delete Success Page Source

You can see the `content_tag` helper generated html for the notice section.

Exercise

Define a class `notice` that will display the text in green. You can add the css to `app/assets/stylesheets/application.css` file.

Summary

In this lesson we learned how to delete a given article. We also learned about flash notice to provide user feedback. In the next lesson we will learn about eliminating duplication in views.

9. View Duplication

Objective

- Learn how to eliminate duplication in views by using partials

Steps

Step 1

Look at the `app/views/new.html.erb` and `app/views/edit.html.erb`. There is duplication.

Step 2

Create a file called `_form.html.erb` under `app/views/articles` directory with the following contents:

```
<%= form_for @article do |f| %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :description %><br>
    <%= f.text_area :description %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Step 3

Edit the app/views/articles/new.html.erb and change the content as follows:

```
<h1>New Article</h1>

<%= render 'form' %>
```

Step 4

Edit the app/views/articles/edit.html.erb and change the content as follows:

```
<h1>Edit Article</h1>

<%= render 'form' %>
```

Step 5

Go to `http://localhost:3000/articles` and create new article and edit existing article. The name of the partial begins with an underscore, when you include the partial by using the render helper you don't include the underscore. This is the Rails convention for using partials.

If you get the following error:

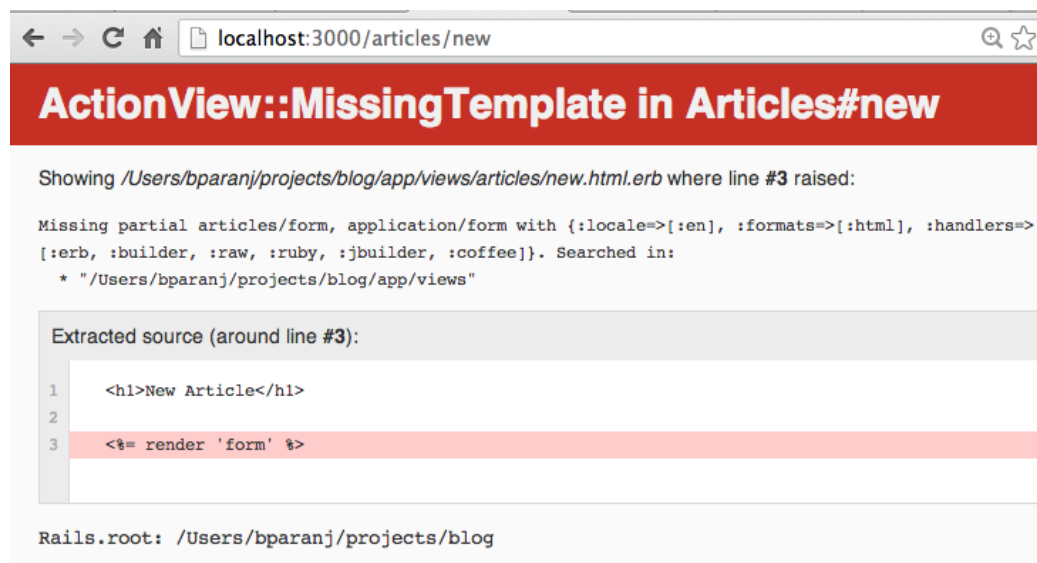


Figure 79: Missing Partial Error

It means you did not create the `app/views/articles/_form.html.erb` file. Make sure you followed the instruction in step 2.

Summary

In this lesson we saw how to eliminate duplication in views by using partials. In the next lesson we will learn about relationships between models.

10. Relationships

Objective

- To learn relationships between models.

Steps

Step 1

Let's create a comment model by using the Rails generator command:

```
~/projects/blog $ rails g model comment commenter:string description:text article:references
  invoke  active_record
   create  db/migrate/20131111023716_create_comments.rb
   create  app/models/comment.rb
   invoke  test_unit
   create  test/models/comment_test.rb
   create  test/fixtures/comments.yml
~/projects/blog $
```

Figure 80: Generate Comment Model

```
$ rails g model comment commenter:string description:text article:references
```

Step 2

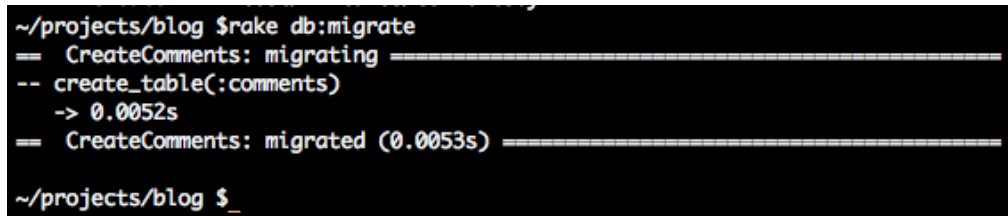
Open the `db/migrate/xyz_create_comments.rb` file in your IDE. You will see the `create_table()` method that takes comments symbol `:comments` as the argument and the description of the columns for the comments table.

What does references do? It creates the foreign key `article_id` in the comments table. We also create an index for this foreign key in order to make the SQL joins faster.

Step 3

Run :

```
$ rake db:migrate
```



```
~/projects/blog $rake db:migrate
== CreateComments: migrating =====
-- create_table(:comments)
   -> 0.0052s
== CreateComments: migrated (0.0053s) =====

~/projects/blog $
```

Figure 81: Create Comments Table

Let's install SQLiteManager Firefox plugin that we can use to open the SQLite database, query, view table structure etc.

Step 4

Install SqliteManager Firefox plugin [SqliteManager Firefox plugin](#)

Let's now see the structure of the comments table.

Step 5

In Firefox go to : Tools -> SQLiteManager

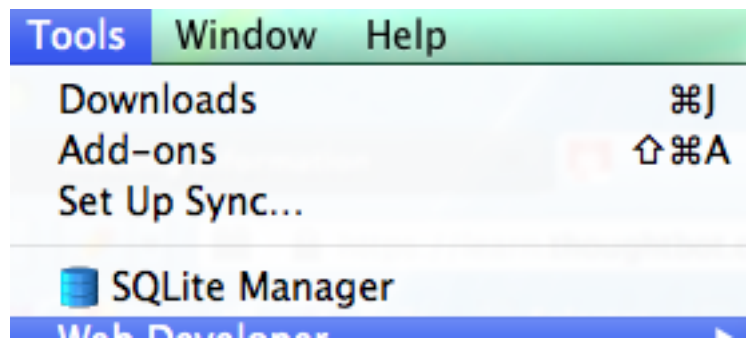


Figure 82: SQLite Manager Firefox Plugin

Step 6

Click on 'Database' in the navigation and select 'Connect Database', browse to blog/db folder.

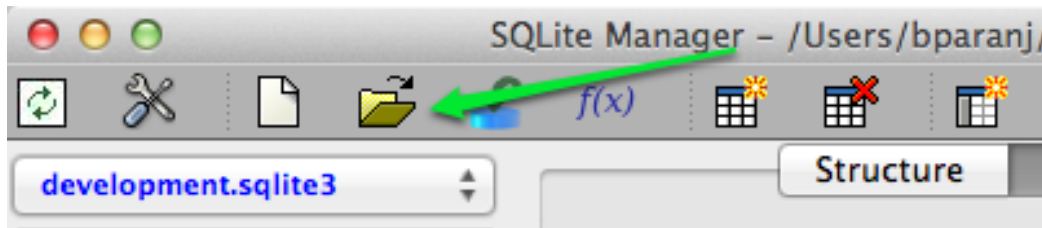


Figure 83: Folder Icon

You can also click on the folder icon as shown in the screenshot.

Step 7

Change the file extensions to all files.

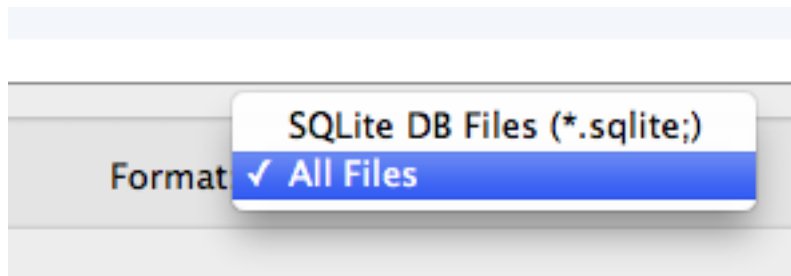


Figure 84: SQLite Manager All Files

Step 8

Open the development.sqlite3 file. Select the comments table.

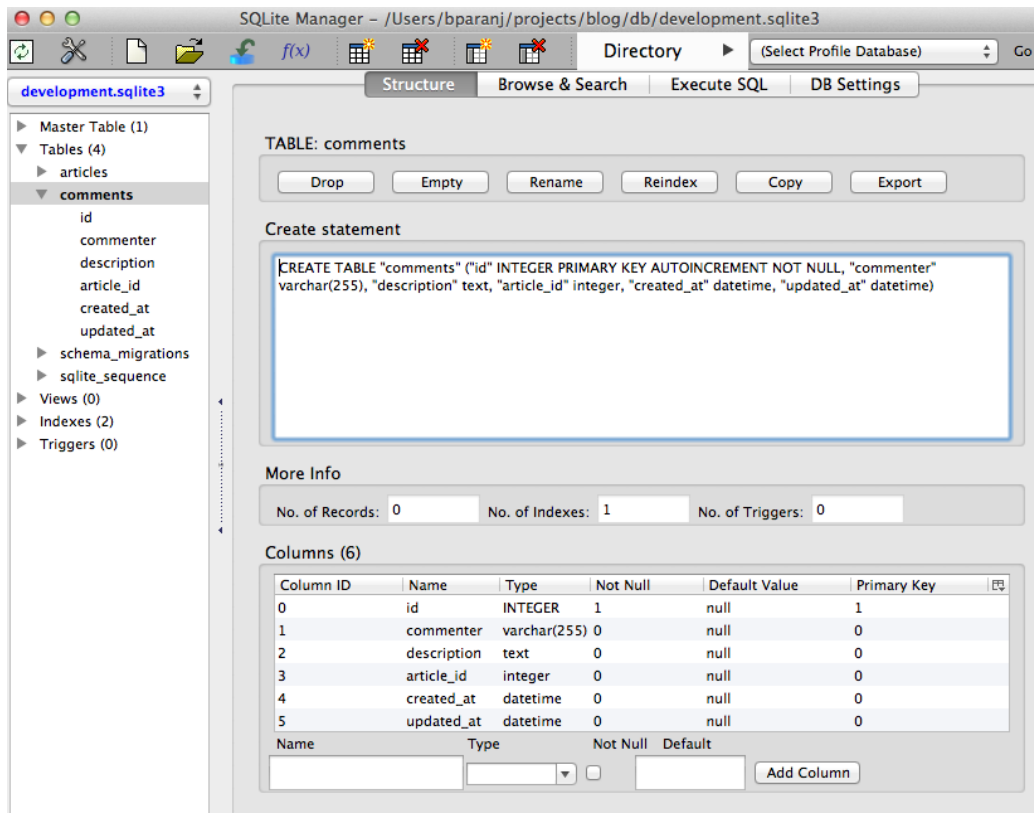


Figure 85: Comments Table Structure

You can see the foreign key `article_id` in the comments table.

Step 9

Open the `app/models/comment.rb` file. You will see the

```
belongs_to :article
```

declaration. This means you have a foreign key `article_id` in the `comments` table.

Step 10

Open the `app/models/article.rb` file. Add the following declaration:

```
has_many :comments
```

This means each article can have many comments. Each comment points to its corresponding article.

Step 11

Open the `config/routes.rb` and define the route for comments:

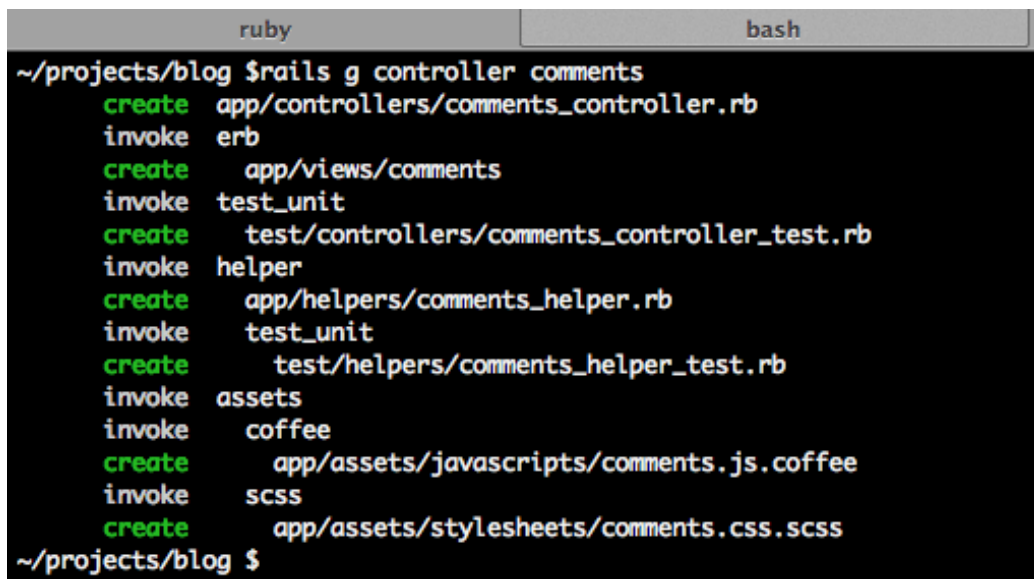
```
resources :articles do
  resources :comments
end
```

Since we have parent-children relationship between articles and comments we have nested routes for comments.

Step 12

Let's create the controller for comments.

```
$ rails g controller comments
```



```
~/projects/blog $ rails g controller comments
create  app/controllers/comments_controller.rb
invoke  erb
create  app/views/comments
invoke  test_unit
create  test/controllers/comments_controller_test.rb
invoke  helper
create  app/helpers/comments_helper.rb
invoke  test_unit
create  test/helpers/comments_helper_test.rb
invoke  assets
invoke  coffee
create  app/assets/javascripts/comments.js.coffee
invoke  scss
create  app/assets/stylesheets/comments.css.scss
~/projects/blog $
```

Figure 86: Generate Comments Controller

Readers can comment on any article. When someone comments we will display the comments for that article on the article's show page.

Step 13

Let's modify the `app/views/articles/show.html.erb` to let us make a new comment:

```
<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br />
```

```

    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

```

The app/views/show.html.erb file will now look like this:

```

<p>
  <%= @article.title %><br>
</p>

<p>
  <%= @article.description %><br>
</p>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

```

Step 14

Go to <http://localhost:3000/articles> page and click on 'Show' for one of the article.



The screenshot shows a web browser window with the address bar displaying `localhost:3000/articles/3`. The page content includes the text "test" and "tester". Below this is a section titled "Add a comment:" in a large, bold, black serif font. Underneath the title are two form fields: "Commenter" with a single-line text input box, and "Description" with a larger multi-line text input box. At the bottom of the form is a button labeled "Create Comment".

Figure 87: Add Comment Form

You will now see the form for filling out the comment for this specific article.

Step 15

View the page source for the article show page by clicking any of the ‘Show’ link in the articles index page.

```
30
31 <h2>Add a comment:</h2>
32 <form accept-charset="UTF-8" action="/articles/3/comments" class="new_comment"
  id="new_comment" method="post"><div style="margin:0;padding:0;display:inline"><input
  name="utf8" type="hidden" value="&#x2713;" /><input name="authenticity_token" type="hidden"
  value="6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=" /></div>
33   <p>
34     <label for="comment_commenter">Commenter</label><br />
35     <input id="comment_commenter" name="comment[commenter]" type="text" />
36   </p>
37   <p>
38     <label for="comment_description">Description</label><br />
39     <textarea id="comment_description" name="comment[description]">
40   </textarea>
41   </p>
42   <p>
43     <input name="commit" type="submit" value="Create Comment" />
44   </p>
45 </form>
```

Figure 88: Add Comment Page Source

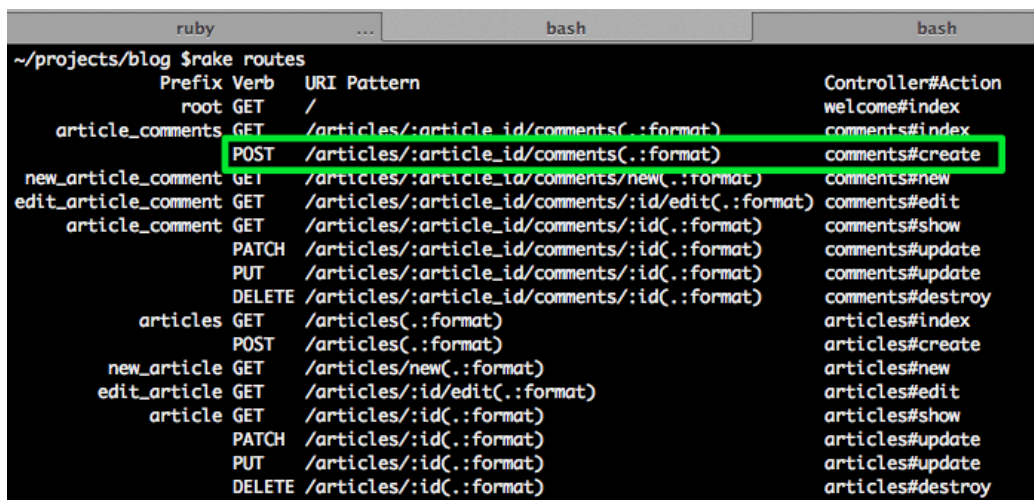
You can see the URI pattern and the http method used when someone submits a comment by clicking the ‘Create Comment’ button.

Exercise 1

Take a look at the output of rake routes and find out the resource endpoint for the URI pattern and http method combination found in step 12.

Step 16

Run rake routes in the blog directory.



```
~/projects/blog $rake routes
      Prefix Verb   URI Pattern
root GET /
article_comments GET /articles/:article_id/comments(:format)
               POST /articles/:article_id/comments(:format)
new_article_comment GET /articles/:article_id/comments/new(:format)
edit_article_comment GET /articles/:article_id/comments/:id/edit(:format)
article_comment GET /articles/:article_id/comments/:id(:format)
                PATCH /articles/:article_id/comments/:id(:format)
                PUT /articles/:article_id/comments/:id(:format)
                DELETE /articles/:article_id/comments/:id(:format)
articles GET /articles(:format)
          POST /articles(:format)
new_article GET /articles/new(:format)
edit_article GET /articles/:id/edit(:format)
article GET /articles/:id(:format)
        PATCH /articles/:id(:format)
        PUT /articles/:id(:format)
        DELETE /articles/:id(:format)
Controller#Action
welcome#index
comments#index
comments#create
comments#new
comments#edit
comments#show
comments#update
comments#update
comments#destroy
articles#index
articles#create
articles#new
articles#edit
articles#show
articles#update
articles#update
articles#destroy
```

Figure 89: Comments Resource Endpoint

You can see how the rails router takes the comment submit form to the comments controller create action.

Step 17

Fill out the comment form and click on ‘Create Comment’. You will get a unknown action create error page.

Step 18

Define the create method in comments controller as follows:

```
def create
```

```
end
```

Step 19

Fill out the comment form and submit it again.

ruby	bash	bash
<pre>Started POST "/articles/3/comments" for 127.0.0.1 at 2013-11-10 19:54:38 -0800 Processing by CommentsController#create as HTML Parameters: {"utf8"=>"✓", "authenticity_token"=>"6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=", "comment"=>{"commenter"=>"bugs", "description"=>"bunny calling earth"}, "commit"=>"Create Comment", "article_id"=>"3"} Completed 500 Internal Server Error in 2ms</pre>		

Figure 90: Comment Values in Server Log

You can see the comment values in the server log.

Step 20

Copy the entire Parameters hash you see from the server log. Go to Rails console and type:

```
params = {"comment"=>{"commenter"=>"test", "description"=>"tester"},
"commit"=>"Create Comment", "article_id"=>"5"}
```

ruby	bash	ruby
<pre>2.0.0p247 :002 > params = {"utf8"=>"✓", "authenticity_token"=>"6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=", "comment"=>{"commenter"=>"test", "description"=>"tester"}, "commit"=>"Create Comment", "article_id"=>"5"} => {"utf8"=>"✓", "authenticity_token"=>"6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=", "comment"=>{"commenter"=>"test", "description"=>"tester"}, "commit"=>"Create Comment", "article_id"=>"5"} 2.0.0p247 :003 ></pre>		

Figure 91: Parameters for Comment

Here you initialize the params variable with the hash you copied in the rails server log.

You can find the value for comment model by doing: `params['comment']` in the Rails console


```
2.0.0p247 :003 > params['comment']  
=> {"commenter"=>"test", "description"=>"tester"}  
2.0.0p247 :004 > _
```

Figure 92: Retrieving Comment

Step 21

Let's create a comment for a given article by changing the create action as follows:

```
def create  
  @article = Article.find(params[:article_id])  
  permitted_columns = params[:comment].permit(:commenter, :description)  
  @comment = @article.comments.create(permitted_columns)  
  
  redirect_to article_path(@article)  
end
```

The only new thing in the above code is the

```
@article.comments.create.
```

Since we have the declaration

```
has_many :comments
```

in the article model. We can navigate from an instance of article to a collection of comments:

```
@article.comments
```

We call the method create on the comments collection like this:

```
@article.comments.create
```

This will automatically populate the foreign key `article_id` in the `comments` table for us.

The `params[:comment]` will retrieve the comment column values.

Step 22

Fill out the comment form and submit it.

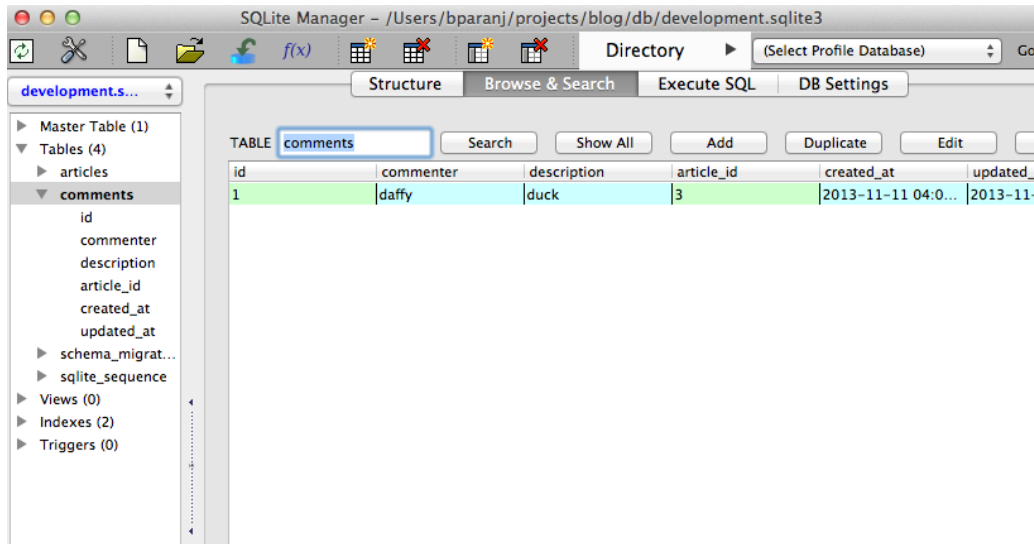


Figure 93: Comment Record in Database

You can now view the record in the MySQLite Manager or Rails dbconsole. Let's now display the comments made for a article in the articles show page.

Step 23

Add the following code to the app/views/articles/show.html.erb

```
<h2>Comments</h2>
<% @article.comments.each do |comment| %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

  <p>
    <strong>Comment:</strong>
```

```

    <%= comment.description %>
  </p>
<% end %>

```

Your app/views/articles/show.html.erb will now look like this:

```

<p>
  <%= @article.title %><br>
</p>

<p>
  <%= @article.description %><br>
</p>

<h2>Comments</h2>
<% @article.comments.each do |comment| %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

  <p>
    <strong>Comment:</strong>
    <%= comment.description %>
  </p>
<% end %>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </p>

```

```
<p>  
  <%= f.submit %>  
</p>  
<% end %>
```

Step 24

Reload the article show page or click on the ‘Show’ link for the article with comments by going to the articles index page.

You will now see the existing comments for an article.

←

→

↺

🏠

📄 localhost:3000/articles/3

test

tester

Comments

Commenter: daffy

Comment: duck

Commenter: bugs

Comment: bunny

Add a comment:

Commenter

Description

Create Comment

Figure 94: Comments For an Article

Summary

We saw how to create parent-child relationship in the database and how to use ActiveRecord declarations in models. We learned about nested routes and how to make forms work in the parent-child relationship. In the next lesson we will implement the feature to delete comments to keep our blog clean from spam.

11. Delete Comment

Objective

- Learn how to work with nested resources

Steps

Step 1

Let's add 'Delete' link for the comment in `app/views/articles/show.html.erb`. We know the hyperlink text will be 'Delete Comment', so:

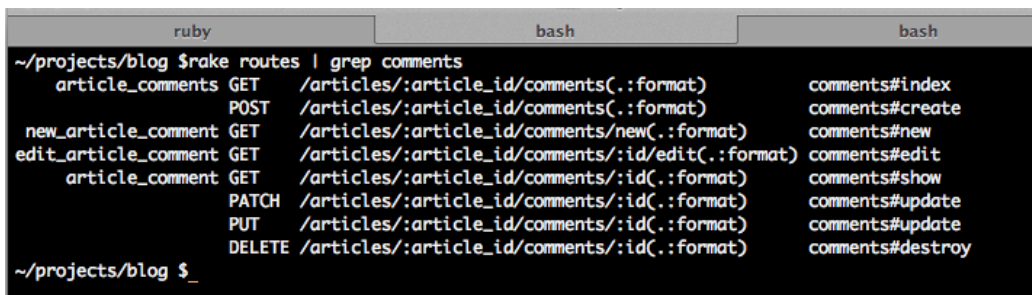
```
<%= link_to 'Delete Comment', ? %>
```

What should be URL helper to use in the second parameter?

Step 2

From the blog directory run:

```
$ rake routes | grep comments
```



```
~/projects/blog $ rake routes | grep comments
  article_comments GET    /articles/:article_id/comments(.:format)  comments#index
                  POST   /articles/:article_id/comments(.:format)  comments#create
  new_article_comment GET    /articles/:article_id/comments/new(.:format) comments#new
  edit_article_comment GET    /articles/:article_id/comments/:id/edit(.:format) comments#edit
  article_comment GET    /articles/:article_id/comments/:id(.:format) comments#show
                  PATCH  /articles/:article_id/comments/:id(.:format) comments#update
                  PUT    /articles/:article_id/comments/:id(.:format) comments#update
                  DELETE /articles/:article_id/comments/:id(.:format) comments#destroy
~/projects/blog $
```

Figure 95: Filtered Routes

We are filtering the routes only to the nested routes so that it is easier to read the output in the terminal.

Step 3

The Prefix column here is blank for the comments controller destroy action. So we go up and look for the very first non blank value in the Prefix column and find the URL helper for delete comment feature.

ruby	bash	bash
~/projects/blog \$rake routes grep comments		
article_comments	GET /articles/:article_id/comments(:format)	comments#index
	POST /articles/:article_id/comments(:format)	comments#create
new_article_comment	GET /articles/:article_id/comments/new(:format)	comments#new
edit_article_comment	GET /articles/:article_id/comments/:id/edit(:format)	comments#edit
article_comment	GET /articles/:article_id/comments/:id(:format)	comments#show
	PATCH /articles/:article_id/comments/:id(:format)	comments#update
	PUT /articles/:article_id/comments/:id(:format)	comments#update
	DELETE /articles/:article_id/comments/:id(:format)	comments#destroy
~/projects/blog \$		

Figure 96: Delete URL Helper for Nested Routes

So, we now have:

```
<%= link_to 'Delete Comment', article_comment(article, comment) %>
```

ruby	bash	bash
~/projects/blog \$rake routes grep comments		
article_comments	GET /articles/:article_id/comments(:format)	comments#index
	POST /articles/:article_id/comments(:format)	comments#create
new_article_comment	GET /articles/:article_id/comments/new(:format)	comments#new
edit_article_comment	GET /articles/:article_id/comments/:id/edit(:format)	comments#edit
article_comment	GET /articles/:article_id/comments/:id(:format)	comments#show
	PATCH /articles/:article_id/comments/:id(:format)	comments#update
	PUT /articles/:article_id/comments/:id(:format)	comments#update
	DELETE /articles/:article_id/comments/:id(:format)	comments#destroy
~/projects/blog \$		

Figure 97: Nested Routes Foreign and Primary Keys

We need to pass two parameters to the URL helper because in the URI pattern column you can see the :article_id as well as the primary key for comment :id. You already know that Rails is intelligent enough to call the id method on the passed in objects. The order in which you pass the objects is the same order in which it appears in the URI pattern.

Step 4

There are other URI patterns which are similar to the comments controller destroy action. So we need to do the same thing we did for articles resource. So the `link_to` now becomes:

```
<%= link_to 'Delete Comment',  
          article_comment(article, comment),  
          method: :delete %>
```

Step 5

The 'Delete Comment' is a destructive operation so let's add the confirmation popup to the `link_to` helper.

```
<%= link_to 'Delete Comment',  
          article_comment(article, comment),  
          method: :delete,  
          data: { confirm: 'Are you sure?' } %>
```

The `app/views/articles/show.html.erb` now looks as follows:

```
<p>  
  <%= @article.title %><br>  
</p>  
  
<p>  
  <%= @article.description %><br>  
</p>  
  
<h2>Comments</h2>  
<% @article.comments.each do |comment| %>  
  <p>  
    <strong>Commenter:</strong>  
    <%= comment.commenter %>  
  </p>
```

```

<p>
  <strong>Comment:</strong>
  <%= comment.description %>
</p>

  <%= link_to 'Delete Comment',
              article_comment_path(article, comment),
              method: :delete,
              data: { confirm: 'Are you sure?' } %>

<% end %>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

```

Step 6

Lets implement the destroy action in the comments controller as follows:

```
def destroy
  @article = Article.find(params[:article_id])
  @comment = @article.comments.find(params[:id])
  @comment.destroy

  redirect_to article_path(@article)
end
```

We first find the parent record which in this case is the article. The next step scopes the find for that particular article record due to security. Then we delete the comment by calling the destroy method. Finally we redirect the user to the articles index page similar to the create action.

Step 7

Go to the articles index page by reloading the `http://localhost:3000/articles`
Click on the ‘Show’ link for any article that has comments.

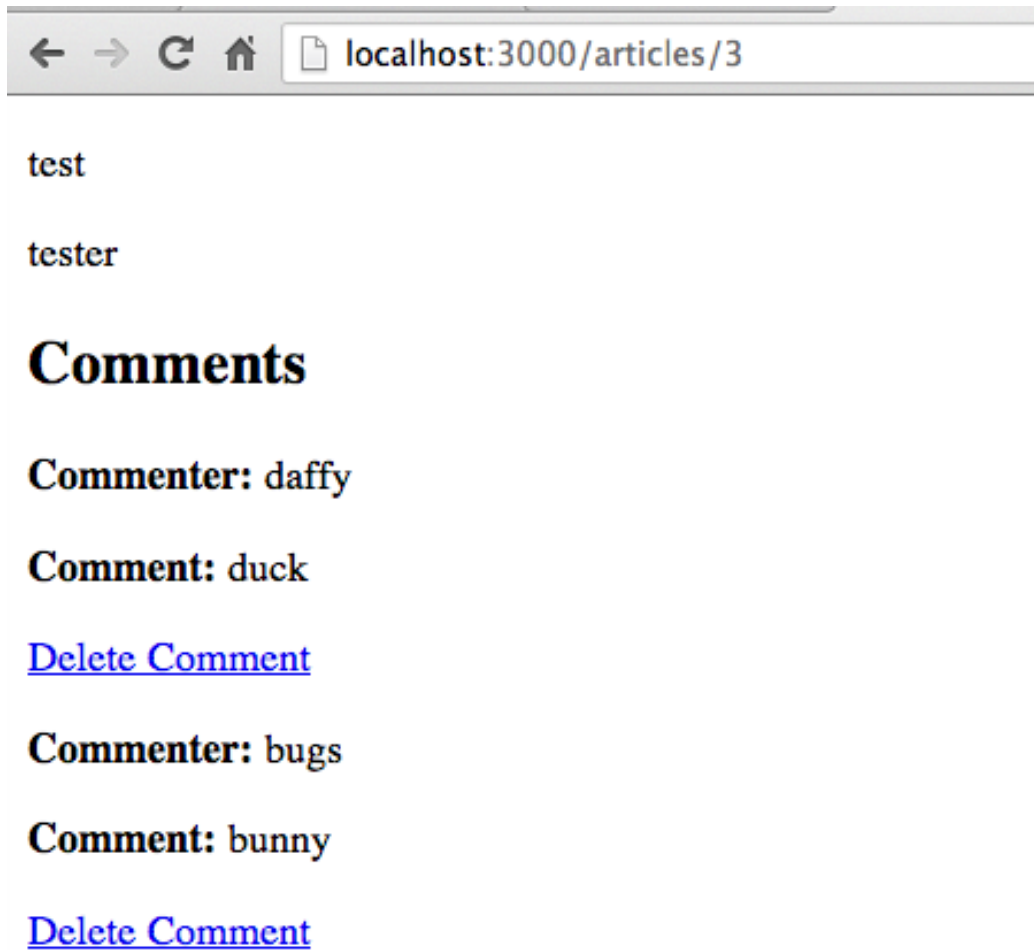


Figure 98: Delete Comment Links

You will see the ‘Delete Comment’ link for every comment of the article.



Figure 99: URL Error

You will get the url error page if you forget to append the `__path` or `__url` to the `article_comment` Prefix.



Figure 100: Article Instance Variable Error

If you forget to use the instance variable @article, then you will get the above error message.

Step 8

Click the 'Delete Comment' link in the articles show page. The confirmation popup will appear and if you click 'Ok' the record will be deleted from the database and you will be redirected back to the articles show page.

Exercise 1

Change the destroy action `redirect_to` method to use notice that says ‘Comment deleted’. If you are using MySQLite Manager you can click on the ‘Refresh’ icon which is the first icon in the top navigation bar to see the comments gets deleted.

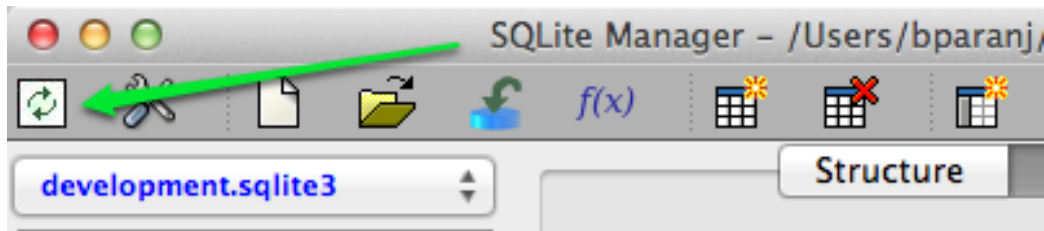


Figure 101: Refresh Icon

Exercise 2

Go to articles index page and delete an article that has comments. No go to either rails dbconsole or use MySQLite Manager to see if the comments associated with that articles is still in the database.

Step 9

When you delete the parent the children do not get deleted automatically. The comment records in our application become useless because they are specific to a given article. In order to delete them when the parent gets deleted we need to change the Article ActiveRecord like this :

```
class Article < ActiveRecord::Base
  has_many :comments, dependent: :destroy
end
```

Now if you delete the parent that has comments, all the comments associated with it will also be deleted. So you will not waste space in the database by retaining records that are no longer needed.

Exercise 3

Change the second parameter, url helper to :

```
[@article, comment]
```

The delete functionality will still work. Since Rails allows passing the parent and child instances in an array instead of using the Prefix.

Summary

In this lesson we learned about nested routes and how to deal with deleting records which has children. Right now anyone is able to delete records, in the next lesson we will restrict the delete functionality only to blog owner.

12. Restricting Operations

Objective

- To learn how to use simple HTTP authentication to restrict access to actions

Steps

Step 1

Add the following code to the top of the `articles_controller.rb`:

```
class ArticlesController < ApplicationController

  http_basic_authenticate_with name: 'welcome',
    password: 'secret',
    except: [:index, :show]

  <!-- actions such as index, new etc omitted here -->
end
```

This declaration protects the creating, editing and deleting functionality. Read only operations such as show and index are not protected.

Step 2

Reload the articles index page : `http://localhost:3000/articles`

Step 3

Click 'Delete' for any of the article.

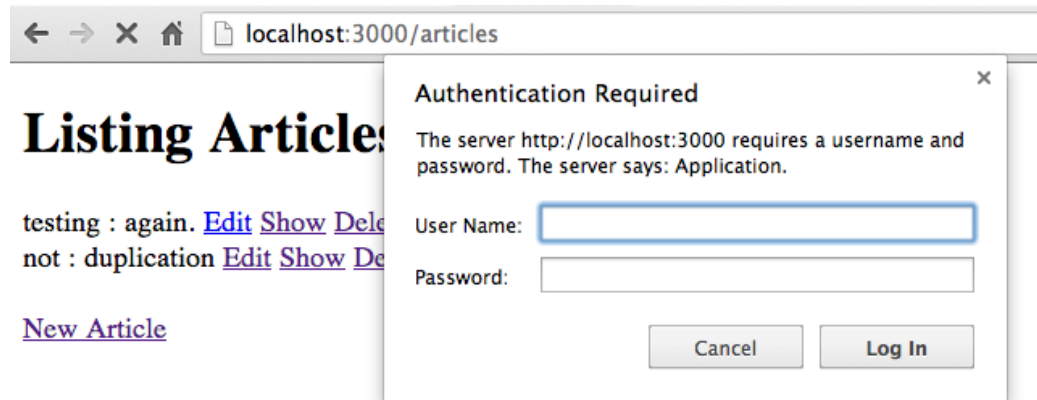


Figure 102: URL Error

You will see popup for authentication.

Step 4

For user name, enter welcome and for password enter secret. Click 'Login'. Now the record will be deleted.

Exercise 1

Use http basic authentication to protect deleting comments in the articles show page.

Summary

This completes our quick tour of Rails 4. If you have developed the blog application following the 12 lessons you will now have a strong foundation to build upon by reading other Rails books to continue your journey to master the Rails framework. Good luck.

Resources

1. [Rails Installer for Windows and Mac OS](#)
2. [Sublime Text 2 IDE](#)
3. [Learning Git](#)

A. Self Learning

Solving Programming Problems

1. Write down your question. This makes you think and clarify your thoughts.
2. Design an experiment to answer that question. Keep the variables to a minimum so that you can solve the problem easily.
3. Run the experiment to learn.

Use the IRB and Rails console to run your experiments.

Learning from Rails Documentation

1. Go to <http://apidock.com/rails>
2. Type the method on the search box at the top.
3. Select the matching result
4. View the documentation, look for an example similar to what you want to accomplish
5. Experiment in the Rails console to learn how it works.
6. Copy it to your project and customize it for your project

Getting Help from Forums

If you have followed the above two suggestions and you still have difficulties, post to forums that clearly explains the problem and what you have done to solve the problem on your own. During this process sometimes you will solve your own problem since explaining the problem to someone will clarify your thinking.

Form Study Group

You can accelerate your learning by forming a study group that meets regularly. If you teach one concept that takes 10 minutes then having a group of 6 people, you can easily cover 6 concepts in one hour.

Practice Makes Perfect

When learning anything new, you will make mistakes. You will go very slow. As you practice you will learn from your mistakes. Learning is a process. Setup 30 mins to an hour everyday for learning. You will get better and faster over time. Repetition is key to gaining development speed.

Survey

Please take the time to answer the three questions below and email them to support@zepho.com . I will review your suggestions and make changes as necessary. Thank you for taking the time to contribute improvements.

1. What did you like about this book?
2. What would you like to see added?
3. What changes should be made and why?