

Rails 4 Quickly

Bala Paranj

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

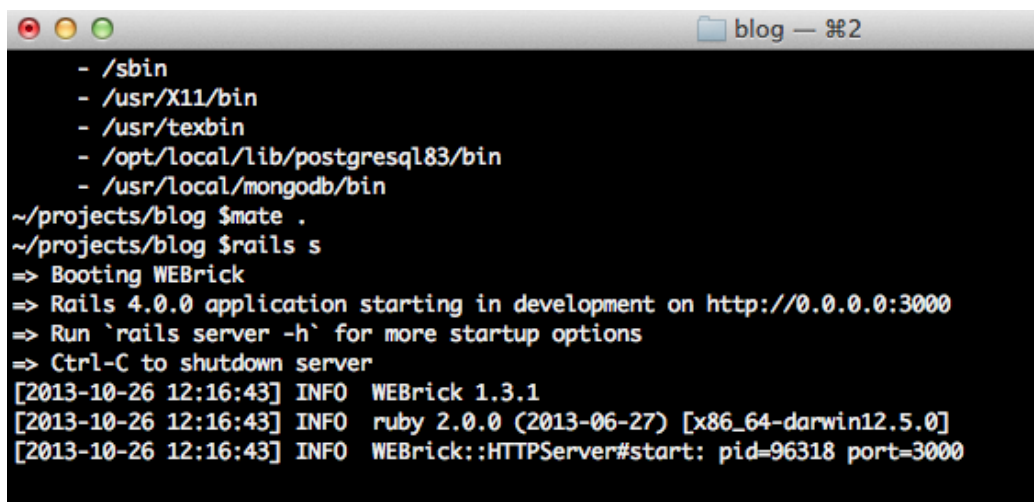
Open the blog project in your favorite IDE. For textmate :

```
$ 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>

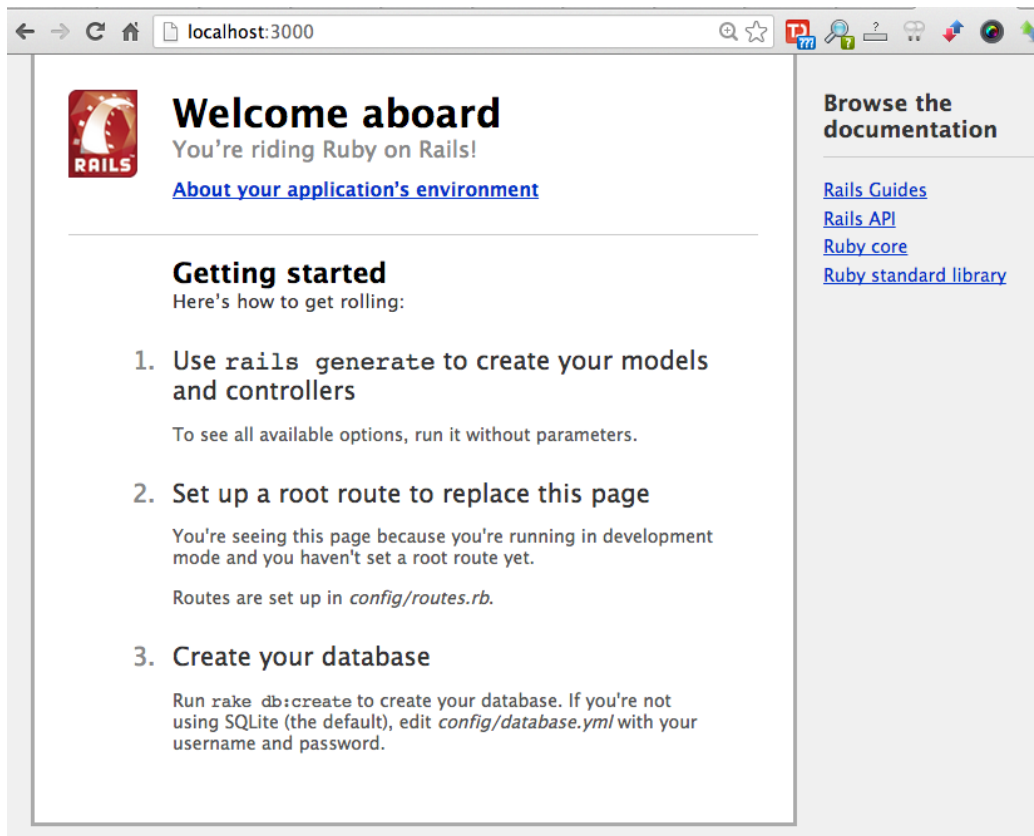


Figure 2: Welcome Aboard

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.

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>

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
```

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

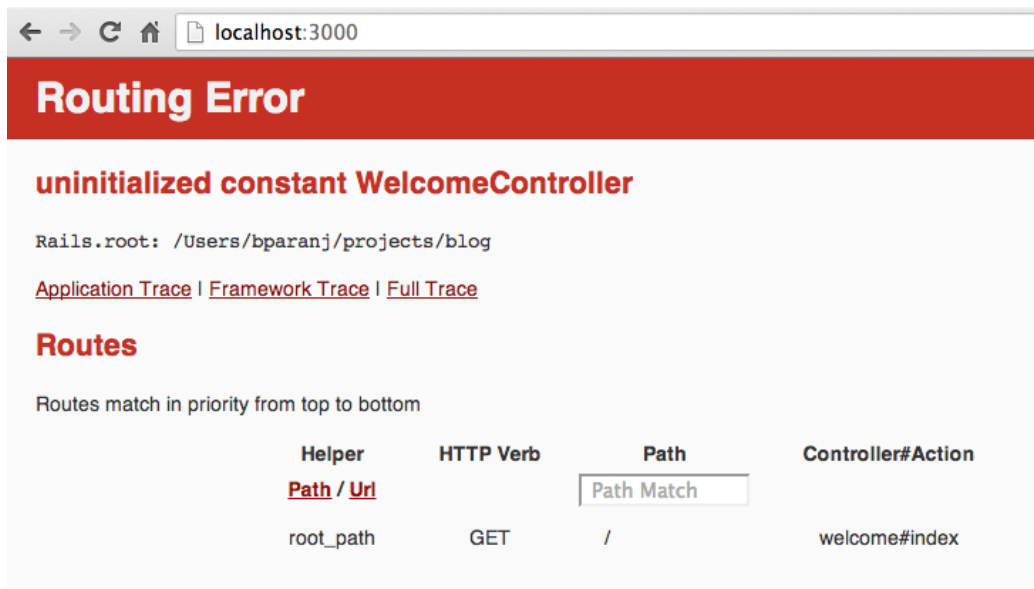


Figure 5: Create Controller

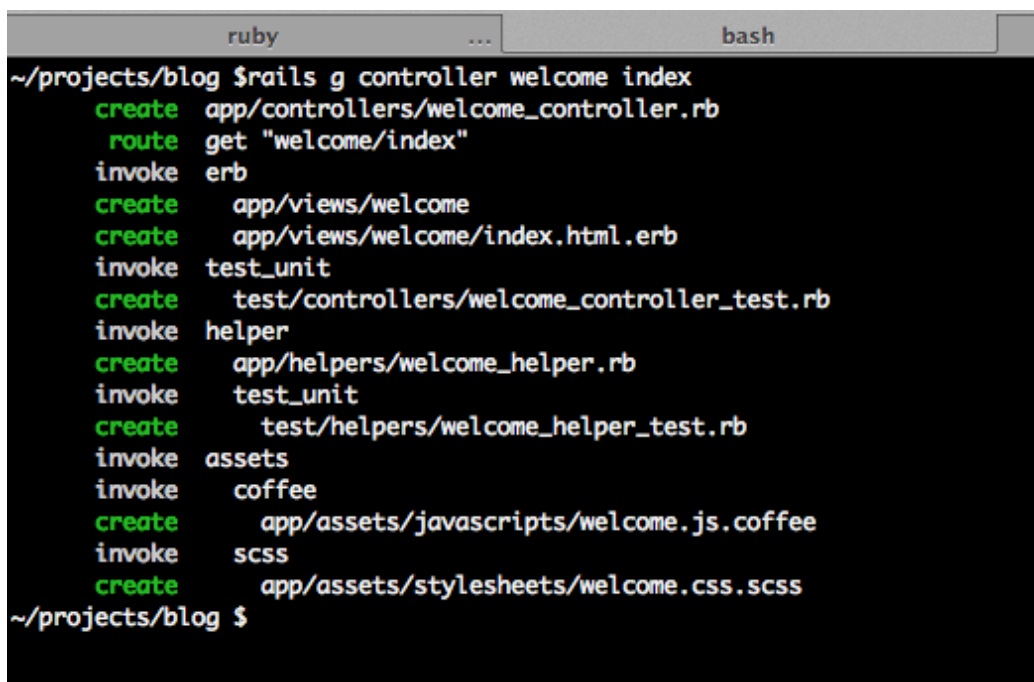


Figure 6: Create Controller

Step 7

Reload the web browser again. You will now see the following page:

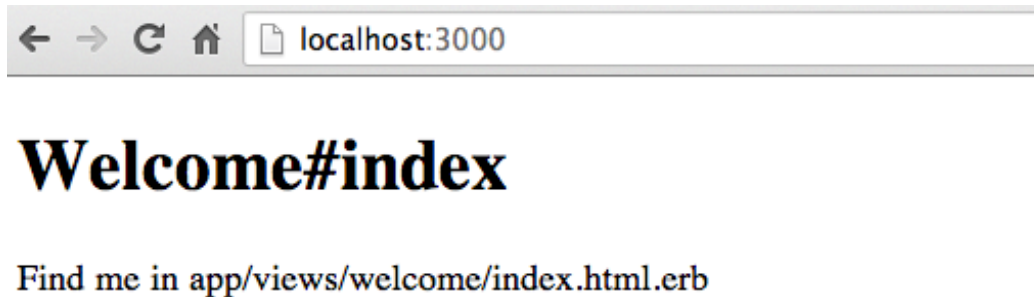


Figure 7: Welcome Page

Step 8

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

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

Save the file.

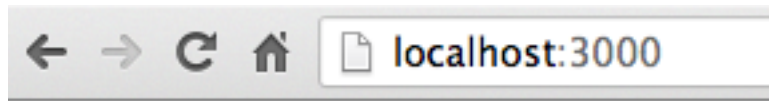
You can embed ruby in `.html.erb` files. In this case we have html only. We will see how to embed ruby in views in the next lesson.

Step 9

Reload the browser. Now you will see 'Hello Rails'.

Step 10

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



Hello Rails

Figure 8: Hello Rails

Step 11

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

ruby	bash
<pre>Started GET "/" for 127.0.0.1 at 2013-10-26 12:40:21 -0700 Processing by WelcomeController#index as HTML Rendered welcome/index.html.erb within layouts/application (0.4ms) Completed 200 OK in 777ms (Views: 775.8ms ActiveRecord: 0.0ms)</pre>	

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. You will now see the error page:

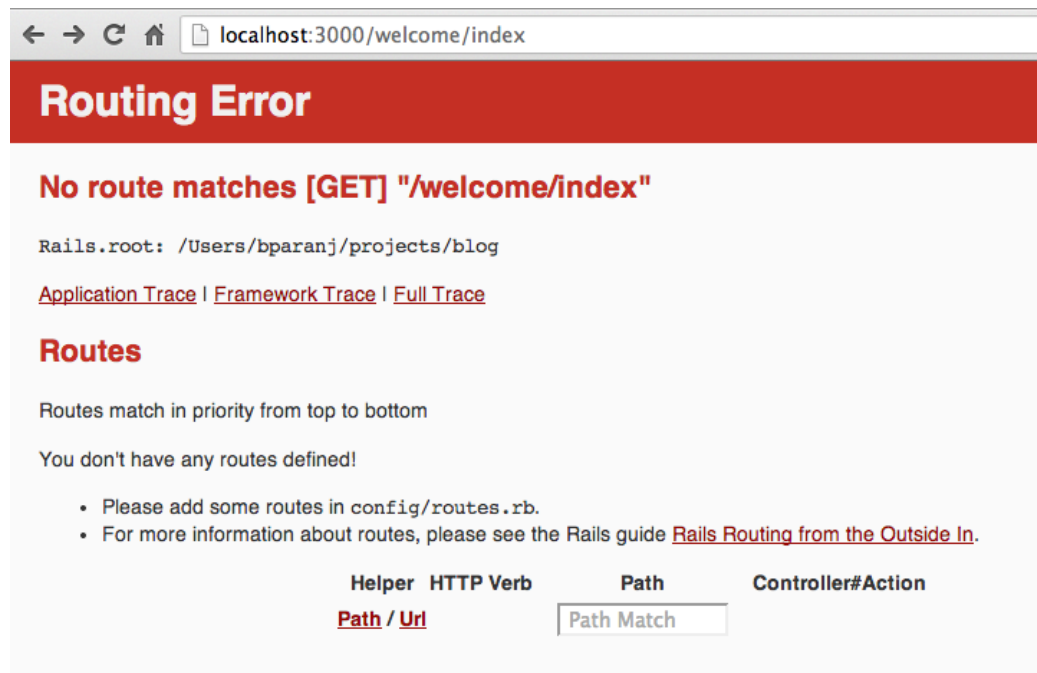


Figure 10: Welcome Index

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.

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
```

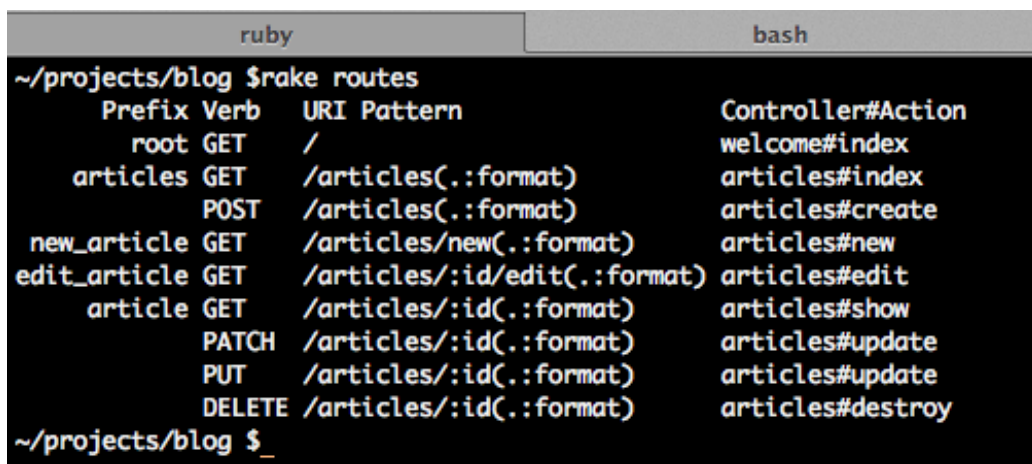
What is a resource? Resource can represent any concept. For instance if you read the documentation for Twitter API <https://dev.twitter.com/docs/api/1.1>, you will see that Timeline is a resource. It is defined in the documentation as collections of Tweets, ordered with the most recent first. There may 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 :

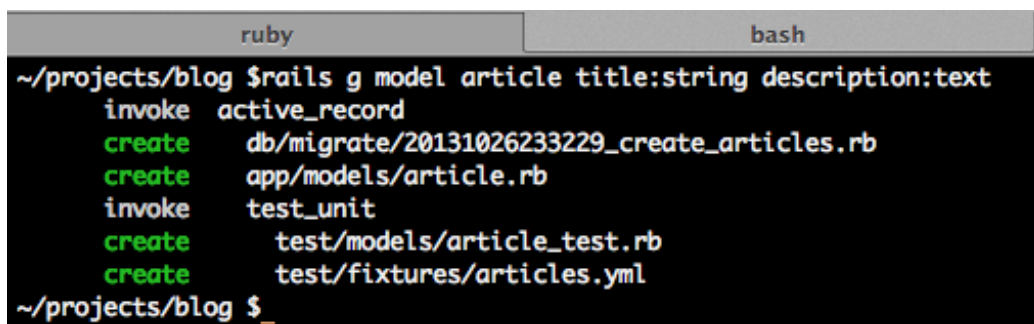

```
resource :article
```

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.

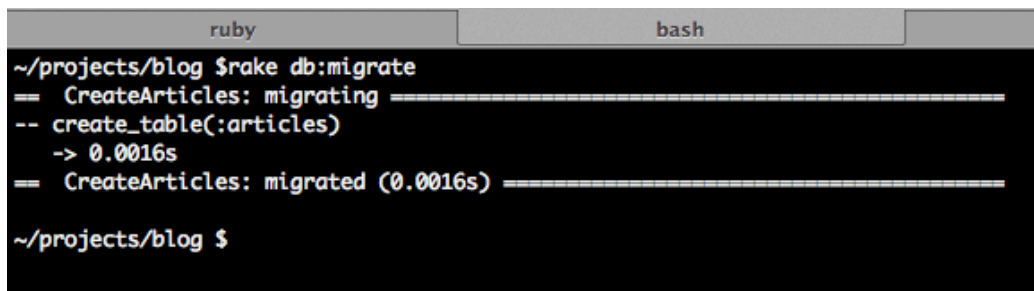
There is a `change()` method in the migration file. Inside the `change()` method there is `create_table()` method that takes the name of the table to create and also the columns and it's data type.

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 text is as follows:

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

~/projects/blog $
```

Figure 13: Create Table

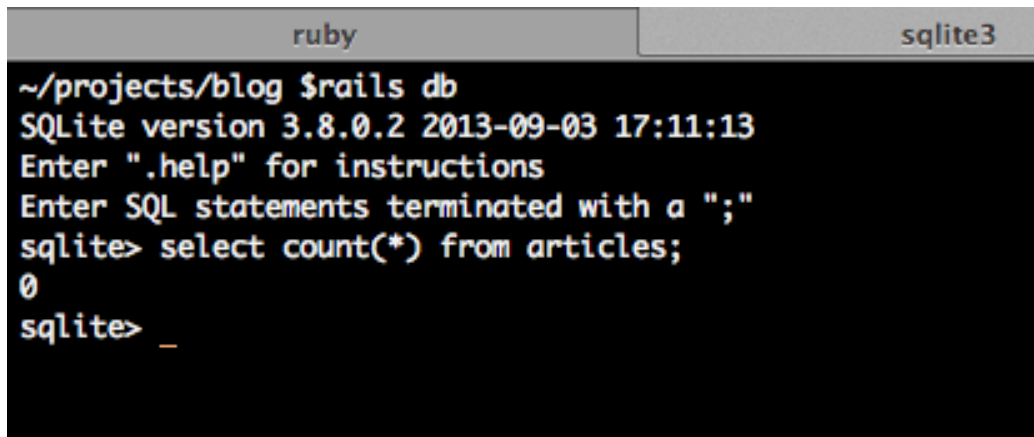
This will create the articles table.

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

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

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

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.

Model View Controller

Objective

- 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 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:

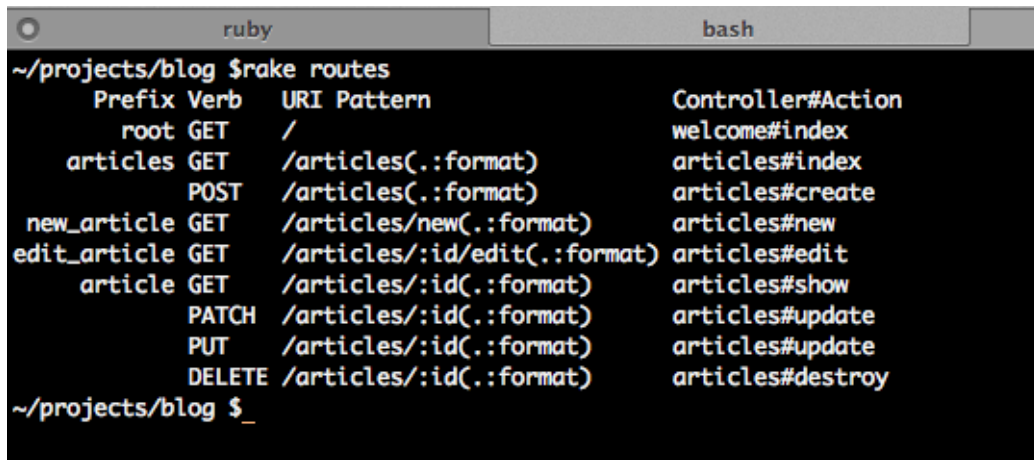
```
<%= link_to 'My Blog', ? %>
```

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.

A terminal window with two tabs: 'ruby' and 'bash'. The 'ruby' tab is active, showing the command '\$rake routes' and its output. The output is a table with columns: Prefix, Verb, URI Pattern, and Controller#Action.

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

The terminal prompt is '~/.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.



Figure 21: My Blog

Step 4

You will find the hyper link in the home page. Right click and do 'View Page Source', 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`. View page source you will see the absolute URL.

Step 6

Click on the 'My Blog' link. You will see the following error page.

Step 7

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



```
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



```
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="http://localhost:3000/articles">My Blog</a>
20 </body>
21 </html>
```

Figure 23: Absolute URL

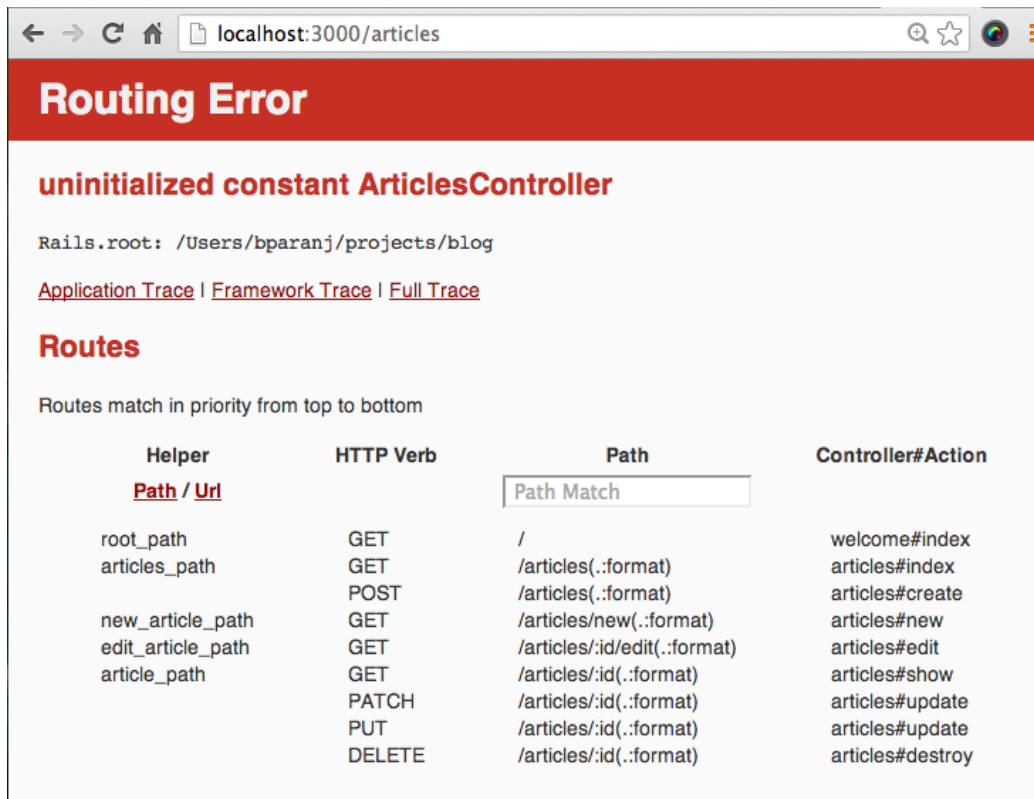


Figure 24: Missing Articles Controller

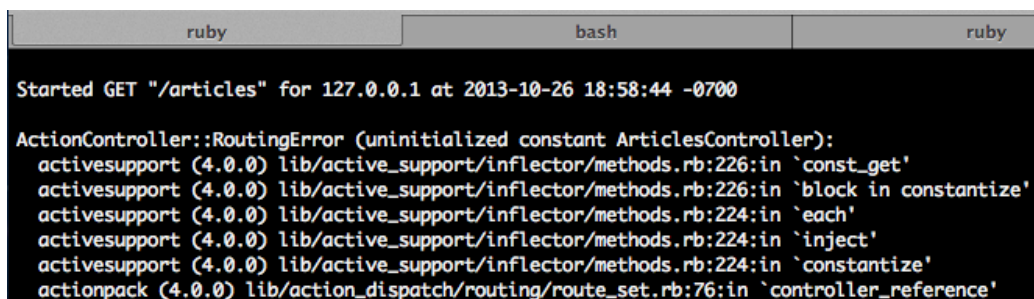


Figure 25: Articles Http Request

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.

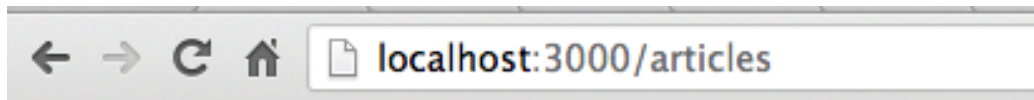
Step 8

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

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

Step 9

Go back to the home page and click on My Blog link. You will see a static page.



Articles#index

Find me in `app/views/articles/index.html.erb`

Figure 26: Articles 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/>

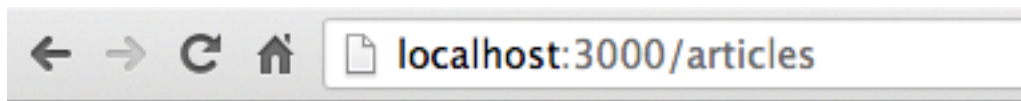
  <%= article.description %>

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



Listing Articles

test
first row another record
different way to create row

Figure 27:

Explanation

View -> Controller -> Model | _____ View

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 was : View -> Controller -> Model -> 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?

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:

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


```
~/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
    article PATCH  /articles/:id(:format)    articles#update
    article PUT    /articles/:id(:format)    articles#update
    article DELETE /articles/:id(:format)    articles#destroy
~/projects/blog $
```

Figure 28: New Article URL Helper



Figure 29: New Article Link

Step 4

Right click on the browser and click ‘View Page Source’, you will see ‘New Article’ link pointing to the resource “/articles/new”.



Figure 30: View Page Source

Step 5

Click the ‘New Article’ link. Go to the terminal and look at the server output. You can see that the browser made a http GET request for the resource “/articles/new” and you see the following error page:

Step 6

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

```
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 31: HTTP Verb Get



Figure 32: Action New Not Found

```
def new
```

```
end
```

Step 7

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



Figure 33: 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. You will now see the following error:

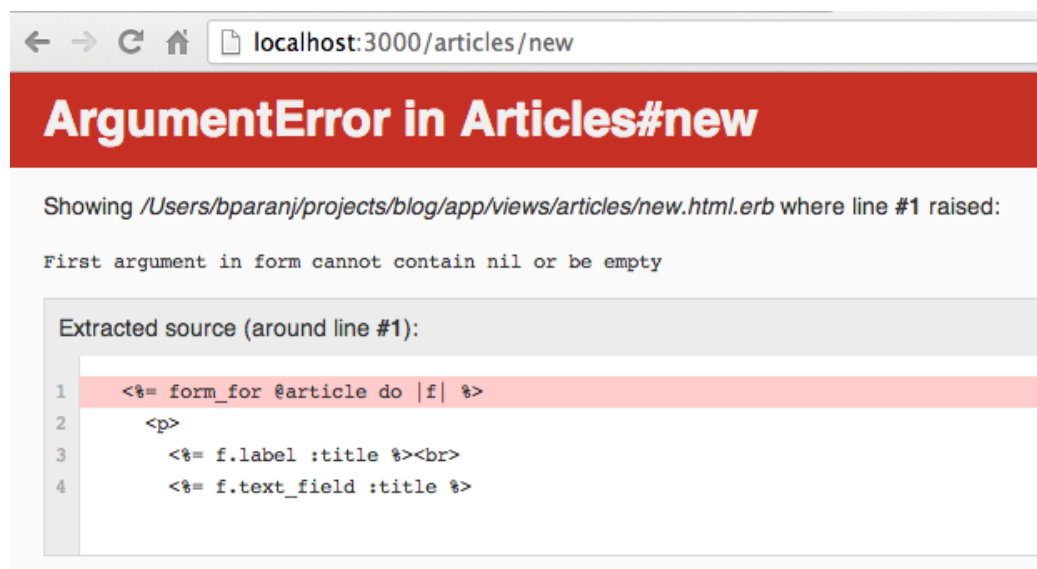


Figure 34: Argument Error

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



The screenshot shows a web browser window with the address bar displaying `localhost:3000/articles/new`. Below the address bar, the form is titled "Title" and "Description". The "Title" field is a single-line text input. The "Description" field is a multi-line text area. At the bottom of the form, there is a button labeled "Save Article".

Figure 35: New Article Form

Step 12

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

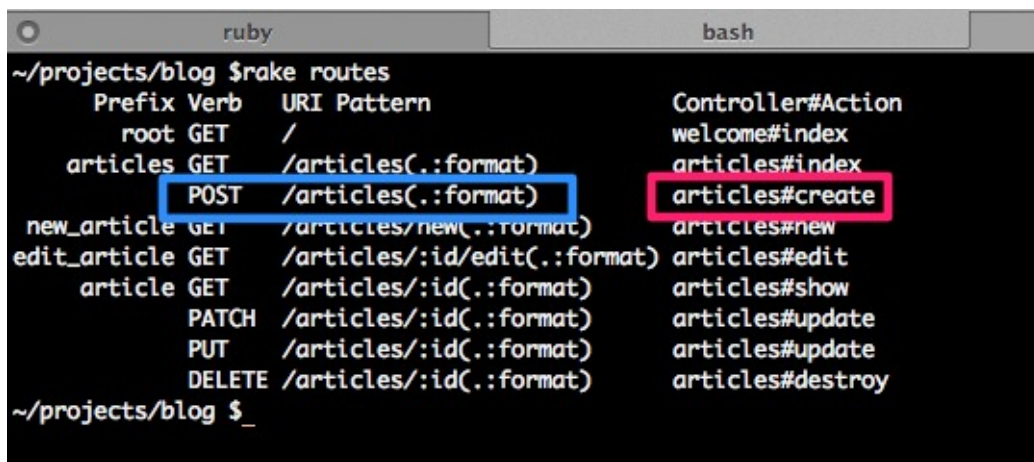
```
17 <body>
18
19 <form accept-charset="UTF-8" action="/articles" class="new_article"
   id="new_article" method="post"><div style="margin:0;padding:0;display:inline">
   <input name="utf8" type="hidden" value="&#x2713;" /><input
   name="authenticity_token" type="hidden"
   value="6gubQ3YqRqyORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=" /></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 36: 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.



```
~/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 37: 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.

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

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:


```
ruby      bash      ruby

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 38: Post Article Server Log



Figure 39: Unknown Action Create

```
def create

end
```

Step 16

Fill out the form and click 'Create Article'.

```
ruby      bash      ruby

Started POST "/articles" for 127.0.0.1 at 2013-11-05 23:42:10 -0800
Processing by ArticlesController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"6gubQ3YqRqYORqwhYYyMiy+NEDkNmGbYcjOPXQg8TBg=", "article"=>{"title"=>"test", "description"=>"tester1"}, "commit"=>"Create Article"}
Completed 500 Internal Server Error in 2ms
```

Figure 40: 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.

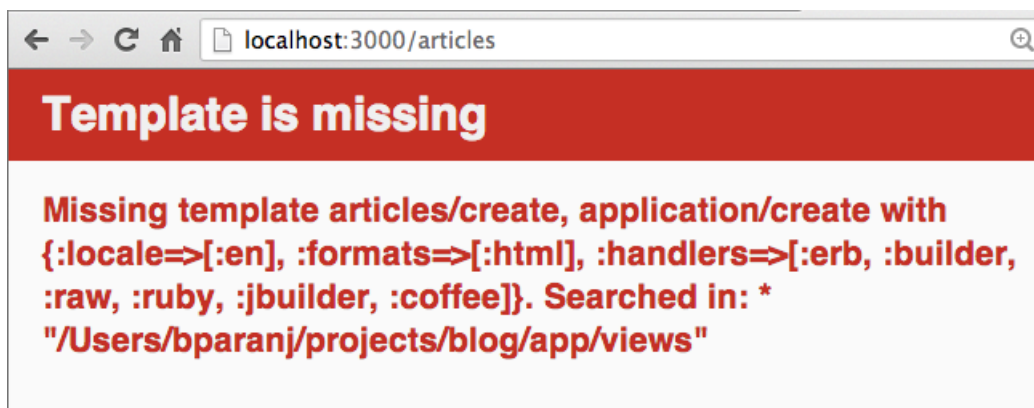


Figure 41: Article Create Missing Template

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. 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:

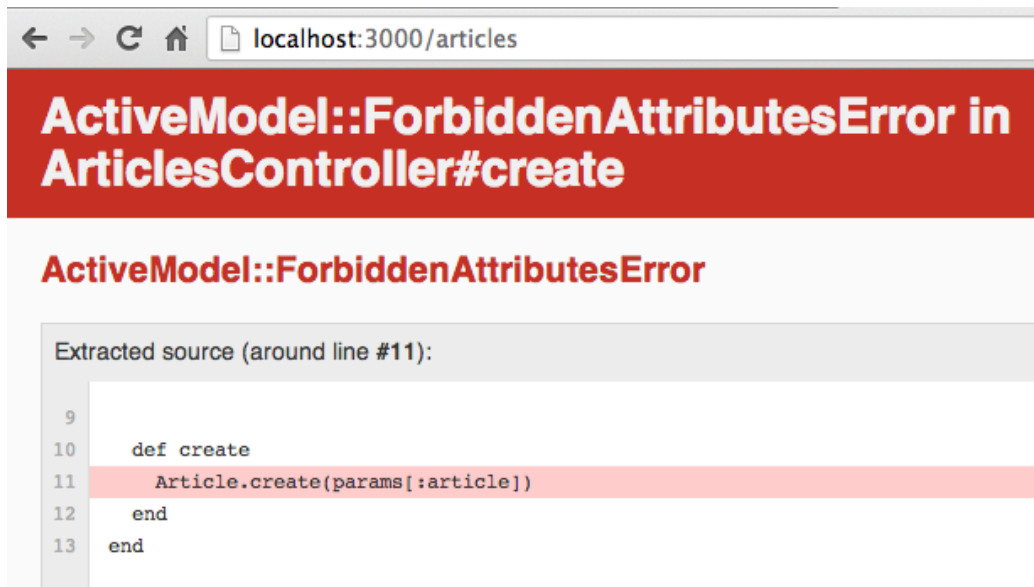


Figure 42: Forbidden Attributes Error

```
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.



Figure 43: 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. You will now see all the articles displayed in the 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.



Figure 44: Displaying All Articles

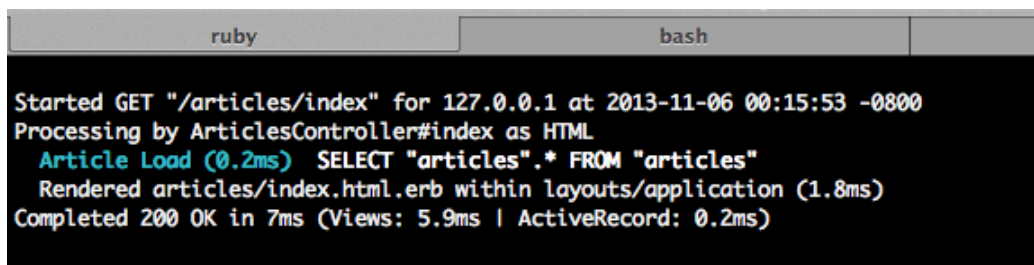


Figure 45: Redirect to Articles Index Page

Update

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 46: 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/>
```

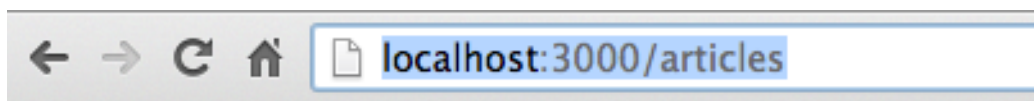
```
<%= link_to 'New Article', new_article_path %>
```

Step 2

Reload the <http://localhost:3000/articles> page. 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', you will see the primary keys of the corresponding row for the :id variable.



Listing Articles

test : first row [Edit](#)

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

test : tester [Edit](#)

testing : again. [Edit](#)

[New Article](#)

Figure 47: Edit Article Link



```
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 48: Edit Article Page Source

Step 4

Click on the 'Edit' link. You will see unknown action edit error page.



Figure 49: Unknown Action Edit

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 50: Argument Error in Articles Edit

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

Step 8

Look at the 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"}
Rendering articles/edit.html.erb within layouts/application (1.4ms)
Completed 500 Internal Server Error in 4ms
```

Figure 51: Edit Article Server Log

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. You will now see the form with values populated.

Step 11

Right click on the browser and click ‘View Page 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.

← → ↻ 🏠 📄 localhost:3000/articles/1/edit

Title

test

Description

first row

Update Article

Figure 52: Edit Article Form

```

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
25 <p>
26 <label for="article_description">Description</label><br>
27 <textarea id="article_description" name="article[description]">
28 first row</textarea>
29 </p>
30
31 <p>
32 <input name="commit" type="submit" value="Update Article" />
33 </p>
34 </form>

```

Figure 53: Edit Article Source

```
ruby bash
~/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
          PATCH  /articles/:id(.:format)         articles#update
          PUT    /articles/:id(.:format)         articles#update
          DELETE /articles/:id(.:format)         articles#destroy
~/projects/blog $
```

Figure 54: 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 that it maps to update action in the articles controller.

```
ruby bash
~/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
          PATCH  /articles/:id(.:format)         articles#update
          PUT    /articles/:id(.:format)         articles#update
          DELETE /articles/:id(.:format)         articles#destroy
~/projects/blog $
```

Figure 55: HTTP Verb PATCH

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])
  @article.update_attributes(params.require(:article).permit(:title, :description))
end
```

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

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

ruby	bash	ruby
<pre>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 ></pre>		

Figure 56: First Article

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])
  @article.update_attributes(params.require(:article).permit(:title, :description))

  redirect_to articles_path
end
```

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

Note : 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.