# 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
```


Figure 1: Rails Server

**Step 7**

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

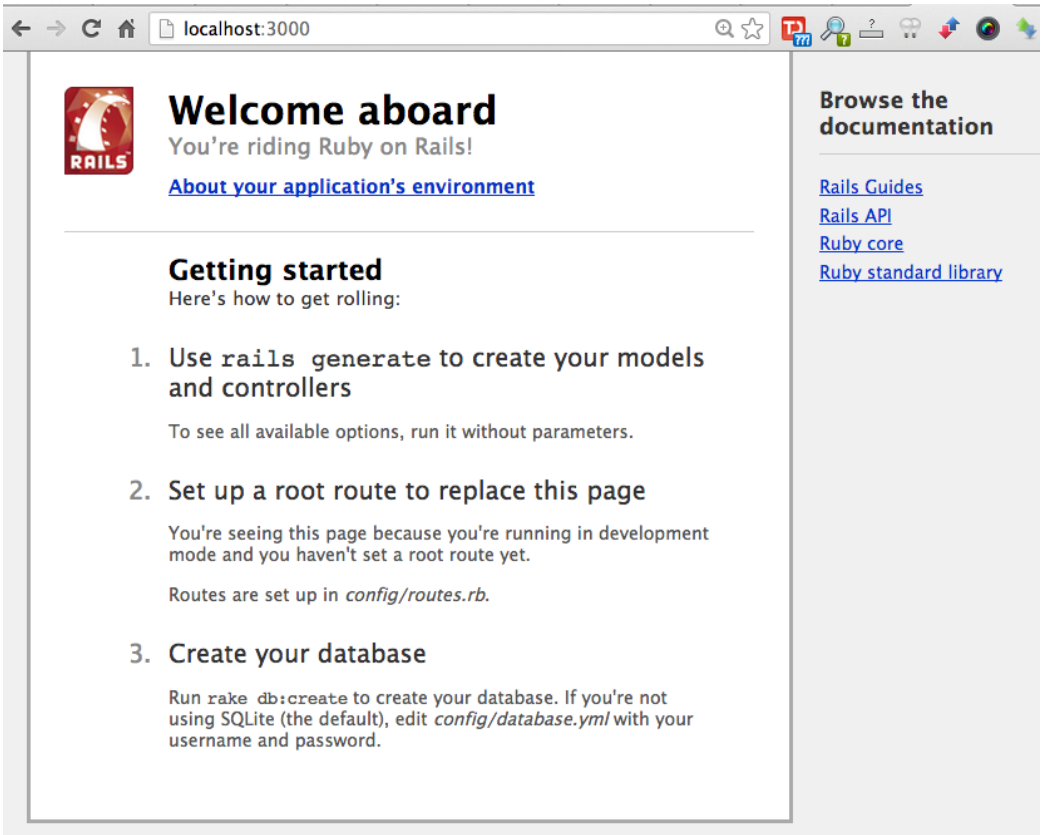


Figure 2: Welcome Aboard

### 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.


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.

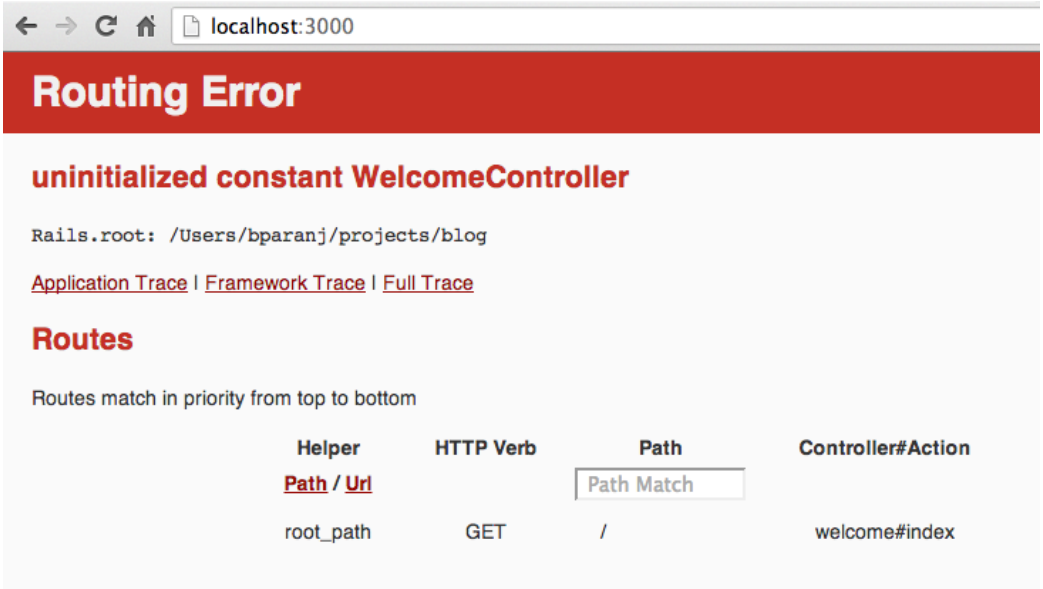


Figure 5: Create Controller

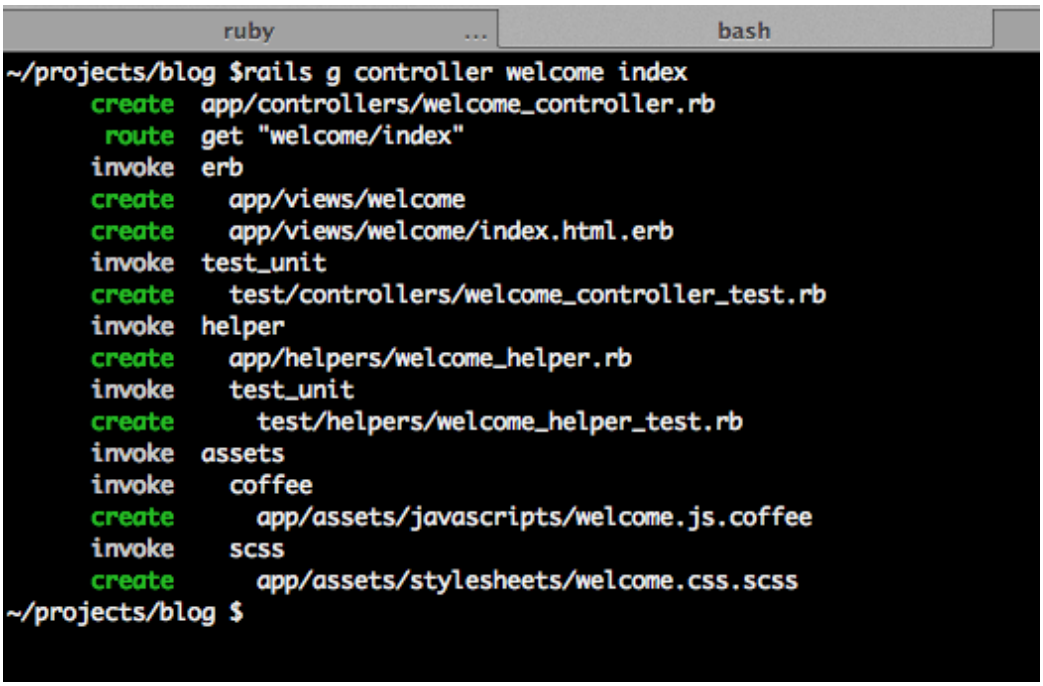


Figure 6: Create Controller

**Step 7**

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

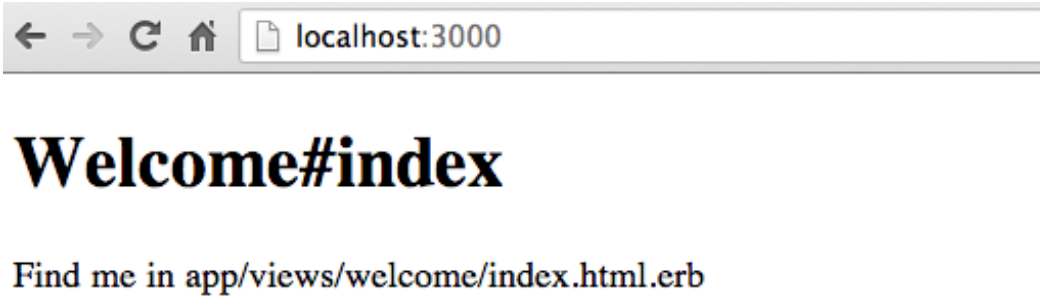


Figure 7: Welcome Page

**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.


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:

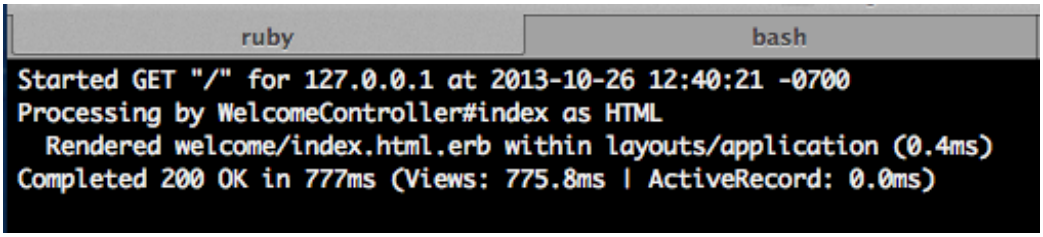


Figure 9: Server Output

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:

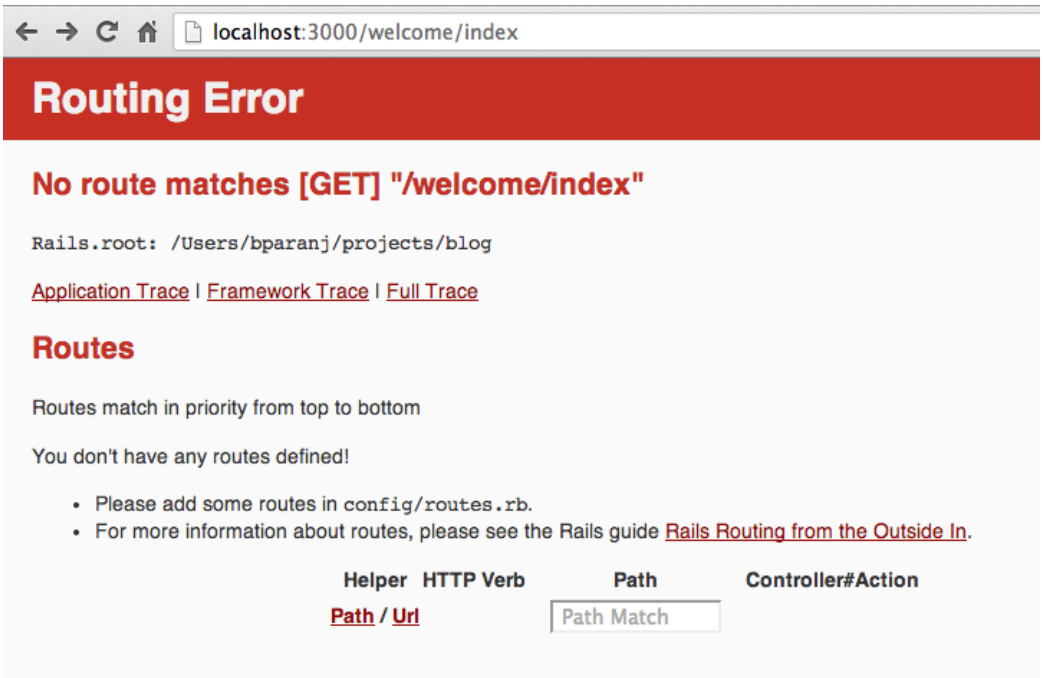


Figure 10: Welcome Index

## 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 documenation for Twitter API https://dev.twitter.com/docs/api/1.1, you will see that Timeline is a resource. It is defined in the documenation 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
```

```
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 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
```

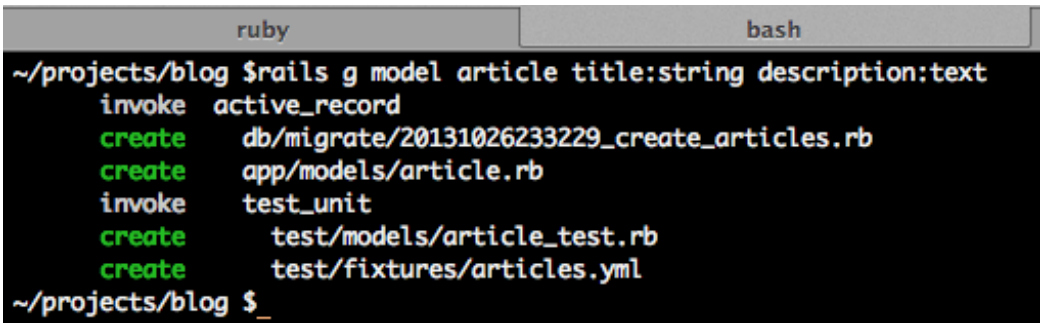


Figure 12: Article Model

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 explictly in the migration file.

### Step 5

Go to the blog directory in the terminal and run :

```
$ rake db:migrate
```

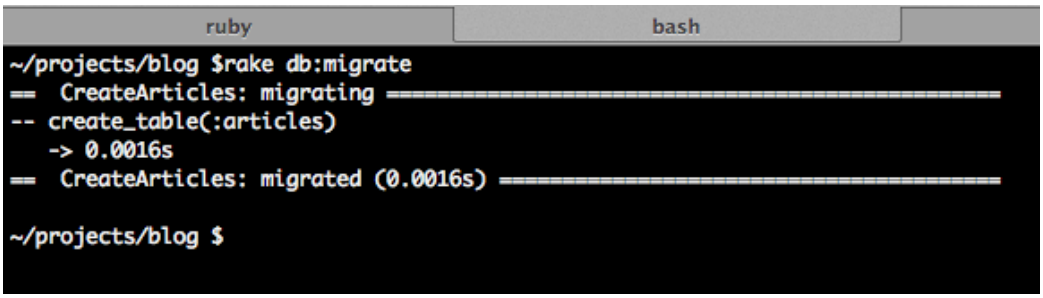


Figure 13: Create Table

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.

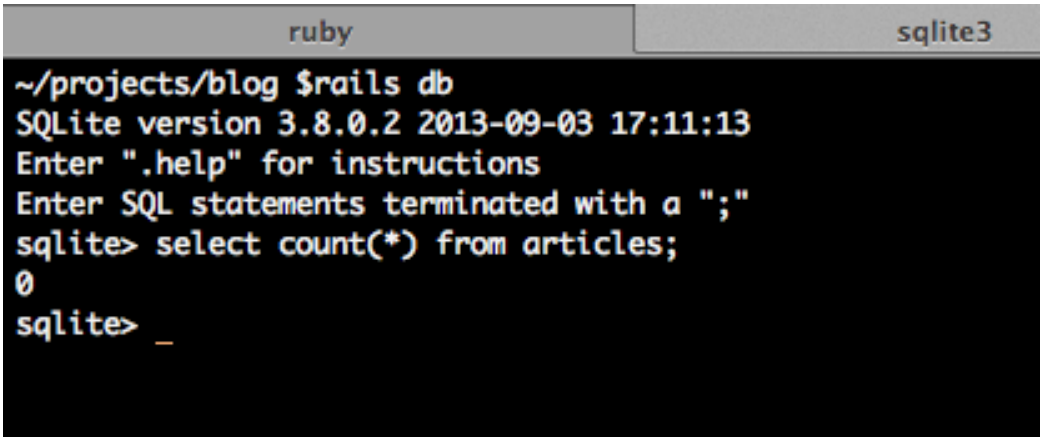


Figure 14: Rails Db Console

**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
~/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 >
```

Figure 15: Rails Console

**Step 10**

Type :

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

```
ruby        sqlite3        ruby
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 >
```

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')
```

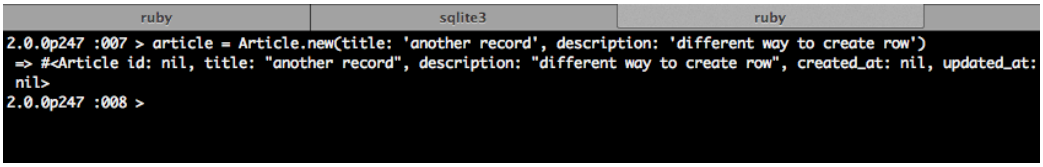


Figure 17: Article Instance

## 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 (?, ?, ?, ?)  [["creat
ed_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.

## 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.

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).

```
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 20: Rake Routes

**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.

**Step 4**

You will the hyper link in the home page. Right click and do 'View Page Source', you will 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.


Figure 21: My Blog

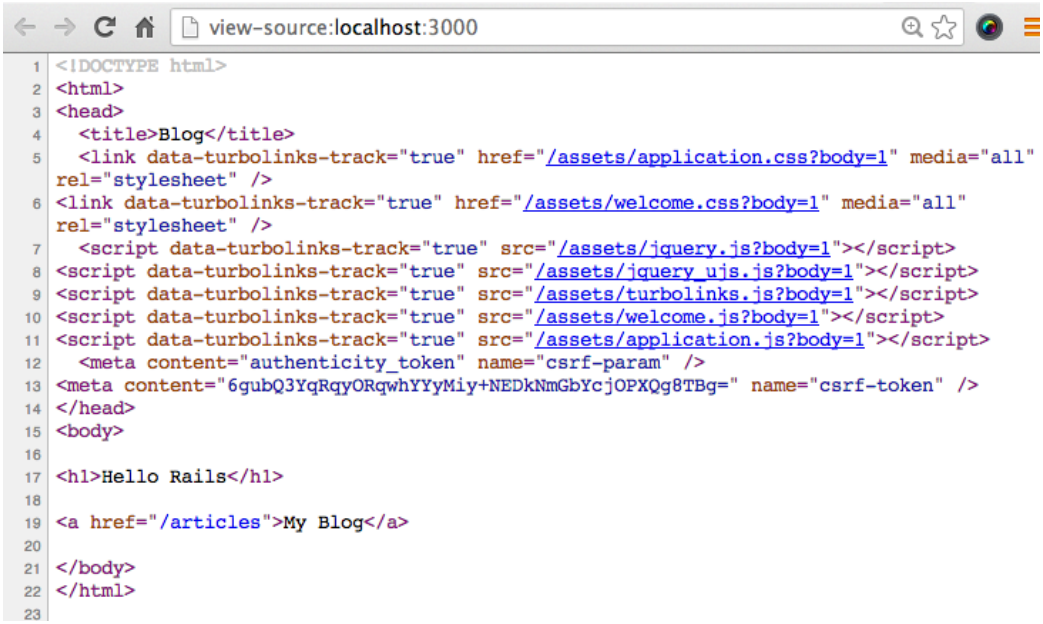


Figure 22: Relative URL

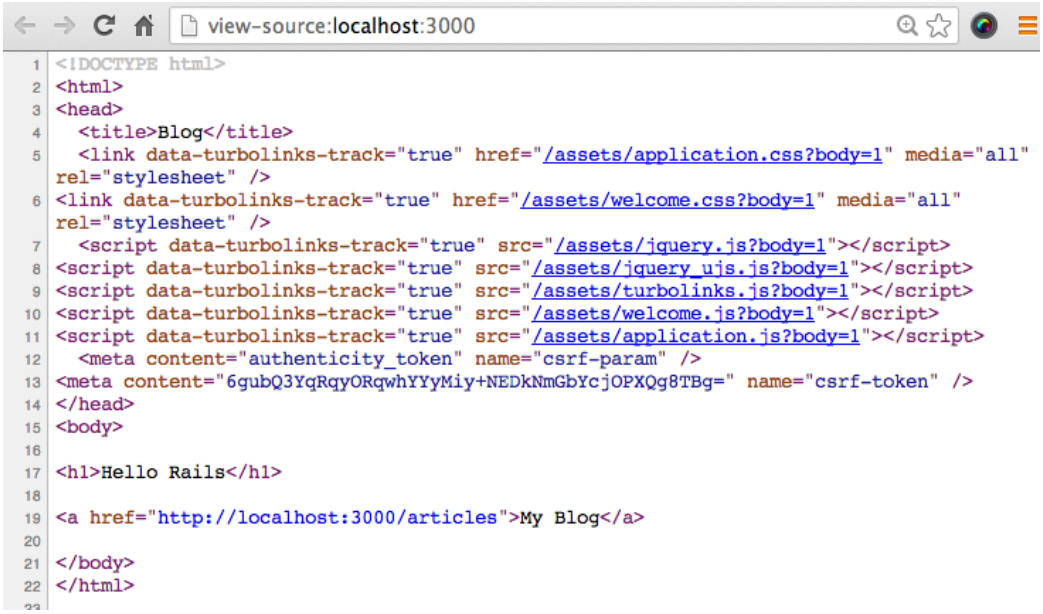


Figure 23: Absolute URL

### 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:

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
```

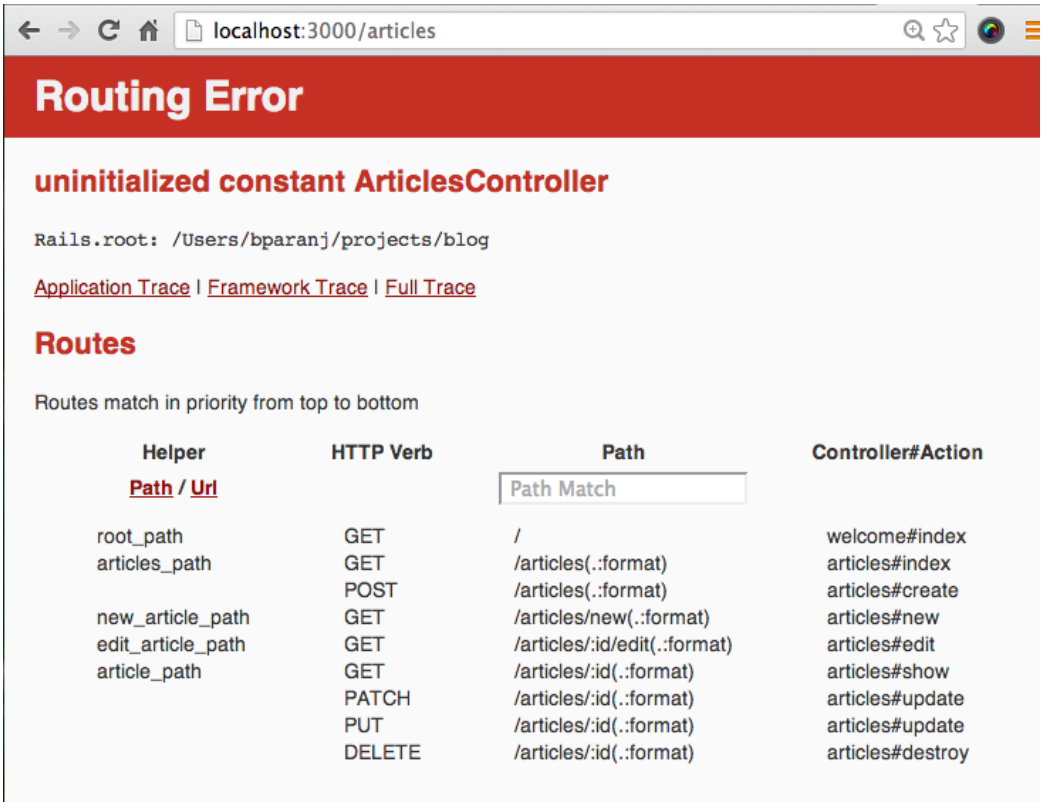


Figure 24: Missing Articles Controller

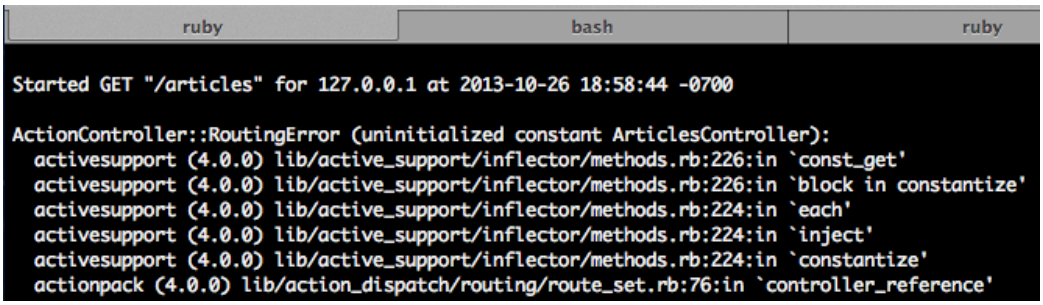


Figure 25: Articles Http Request

**Step 9**

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


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

**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.


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 chapter 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 chapter 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?