

Rails 5 Quickly

Bala Paranj

About the Author

Bala Paranj has a masters degree in Electrical Engineering from The Wichita State University. He has been working in the software industry since 1996. He started his career as Technical Support Engineer and became a Web Developer using Perl, Java and Ruby. He has consulted for companies in USA, Australia and Jamaica in telecommunication, financial, electronic design automation and other domain.

He has professionally worked as a developer using TDD and pair programming for startups. He is the founder of Silicon Valley Ruby Meetup. He has been organizing Ruby, Rails and TDD related events since 2007. He has taught TDD Bootcamps and TDD tutorials for Silicon Valley Ruby Meetup members for more than 3 years. He has spoken at Silicon Valley Ruby Meetup, San Francisco Ruby Meetup and Silicon Valley Code Camp. He is the author of Test Driven Development in Ruby published by Apress. You can reach him at feedback@rubyplus.com with any feedback.

About Reader

This book assumes that you have already installed Ruby 2.2, Rails 5 and your favorite IDE such as Sublime, RubyMine, Textmate etc. The reader must already have a basic understanding of Ruby language. This is a short book. The objective is to bring you up to speed in Rails 5 quickly. Hence the title Rails 5 Quickly. This book is written for beginners who want to learn the fundamentals. It will give you a solid foundation for you to build upon.

The book's main focus is on Rails. You will not find any discussion of Cucumber, Git, Heroku, RSpec, FactoryGirl or any other irrelevant topics. It provides 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.

Acknowledgments

This book is the result of teaching Rails tutorials at the Silicon Valley Ruby meetup. The members of Silicon Valley Ruby meetup provided me early feedback on every chapter. This book is also an experiment in 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 debts to the creator of Ruby, Matz for creating such a beautiful language; as well as the Ruby community for creating useful frameworks and gems to make a developer’s life easy. I hope this book makes your learning process a little easier.

How to Read this Book

This step-by-step tutorial was written as a hands-on guide to Rails. You must read and follow the instructions to work through the application we will be developing. It is written to be read sequentially. Learning by doing is the best way to understand something new. So, make an attempt to do the exercises. This will make your brain more receptive to absorbing the concepts.

Software Versions Used

Ruby Gems : 2.5.1 Ruby : 2.3.1 Rails : 5.0

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 B. Troubleshooting C. FAQ

CHAPTER 1

Running the Server

Objective

- To run your rails application on your machine and check your application's environment.

Steps

Step 1

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

```
$ ruby -v
```

The output on my machine is : ruby 2.3.1p112 (2016-04-26 revision 54768)
[x86_64-darwin11.0]

```
$ rails -v
```

The output on my machine is: Rails 5.0.0

```
ruby $ gem env
```

The output on my machine is: RUBYGEMS VERSION: 2.5.1

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 --skip-spring
```

We are skipping Spring because we don't need it.

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
```

```
=> Booting Puma
=> Rails 5.0.0 application starting in development on http://localhost:3000
=> Run 'rails server -h' for more startup options
Puma starting in single mode...
* Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl Brawl
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```



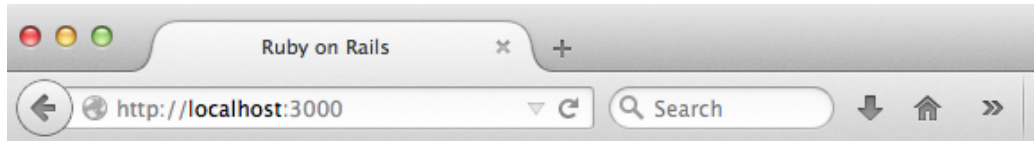
```
zepho-mac-pro:rlog5 zepho$ free -m
-bash: free: command not found
zepho-mac-pro:rlog5 zepho$ rails s
=> Booting Puma
=> Rails 5.0.0 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl Brawl
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

Figure 1: Rails Server

Step 7

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

Welcome page displayed as the home page.



Yay! You're on Rails!



Rails version: 5.0.0

Ruby version: 2.3.1 (x86_64-darwin11.0)

Figure 2: Welcome Aboard

Step 8

You can shutdown your server by pressing Control+C.

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. In the next lesson you will learn how to create a home page for your web application.

CHAPTER 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

Add the line :

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

to the routes.rb. So it now looks like this:

```
Rails.application.routes.draw do
  root 'welcome#index'
end
```

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

Step 3

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

```
rails routes
```

```
Prefix Verb URI Pattern Controller#Action
root GET  /          welcome#index
```

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 `http://localhost:3000`

Step 4

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

We see the uninitialized constant `WelcomeController` routing error. This happens because we don't have a welcome controller to handle the incoming GET request for the home page.

Step 5

You can either open a new terminal and go to the blog project directory or open a new tab in your terminal and go to the blog project directory.

Step 6

In this new tab or the terminal, go the blog project directory and type:

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

You will see the output:

```
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
invoke  assets
invoke  coffee
create  app/assets/javascripts/welcome.coffee
invoke  css
create  app/assets/stylesheets/welcome.css
```

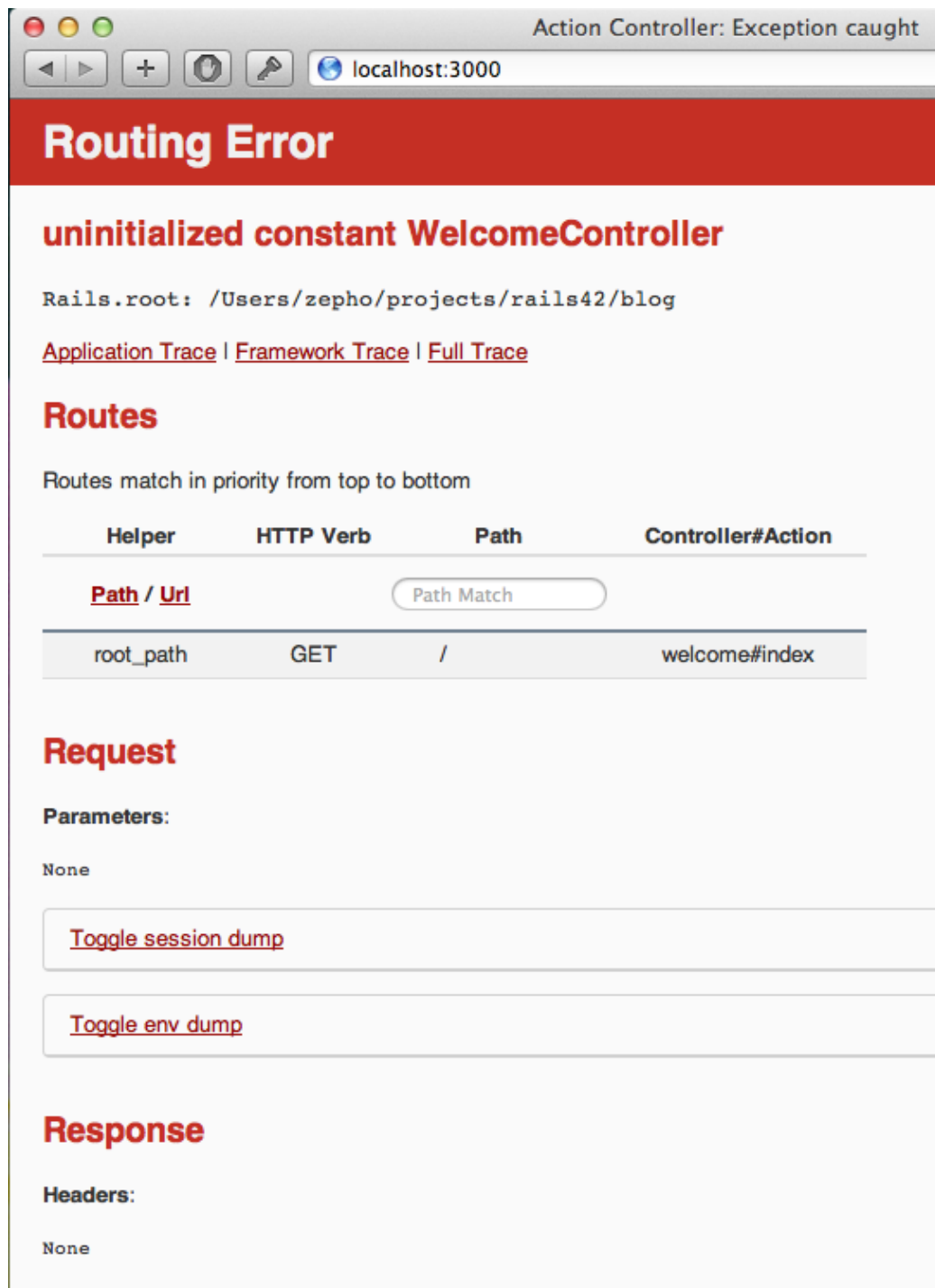


Figure 3: Uninitialized Constant WelcomeController

rails command takes the arguments g for generate, then the controller name and the action. In this case the controller name is welcome and the action name is index.

Step 7

Reload the web browser again.

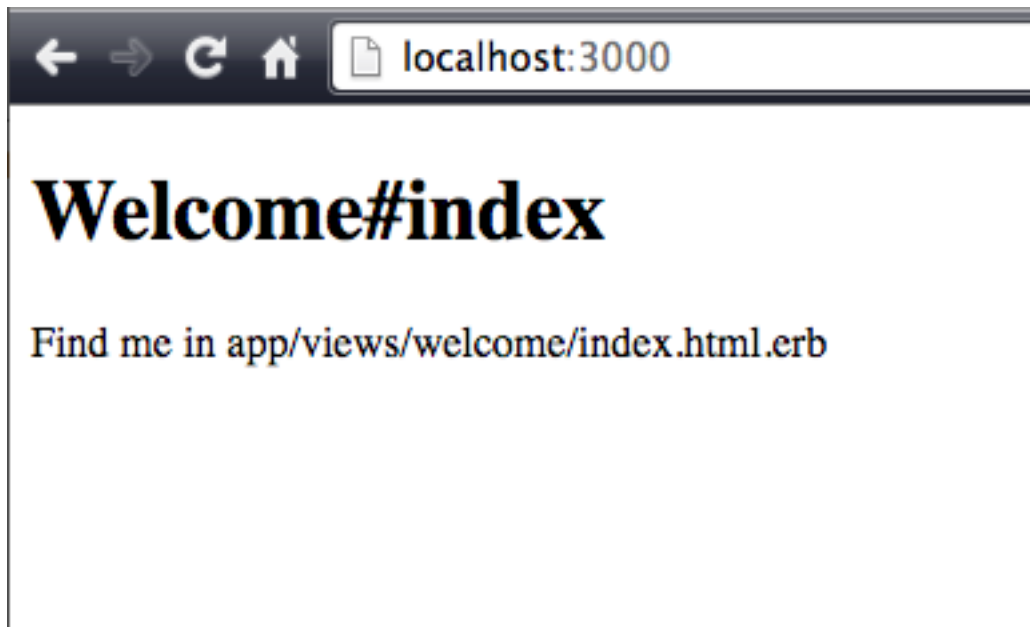


Figure 4: Welcome Page

You will now see the welcome page.

Step 8

Go to `app/views/welcome/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. The `.erb` stands for embedded Ruby. 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.

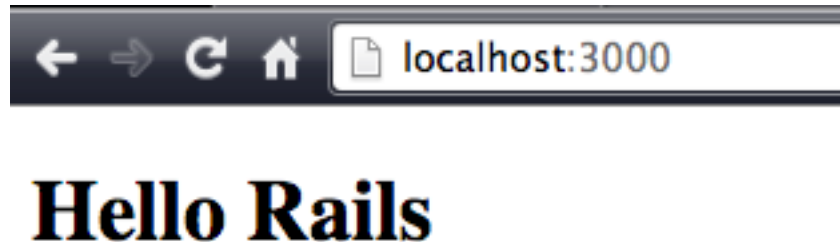


Figure 5: Hello Rails

Now you will see ‘Hello Rails’ as the home page content.

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:

```
Started GET "/" for ::1 at 2016-07-05 15:22:44 -0700
Processing by WelcomeController#index as HTML
  Rendering welcome/index.html.erb within layouts/application
  Rendered welcome/index.html.erb within layouts/application (0.5ms)
Completed 200 OK in 25ms (Views: 22.8ms | ActiveRecord: 0.0ms)
```

```
Use Ctrl-C to stop
Started GET "/" for ::1 at 2016-07-05 15:21:54 -0700
Processing by WelcomeController#index as HTML
  Rendering welcome/index.html.erb within layouts/application
  Rendered welcome/index.html.erb within layouts/application (1.4ms)
Completed 200 OK in 1867ms (Views: 1844.3ms | ActiveRecord: 0.0ms)
```

Figure 6: 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, the view that corresponds to the index action is `app/views/welcome/index.html.erb`.

Step 12

Open a new terminal or a new tab and go to Rails console by running:

```
$rails c
```

from the blog project directory.

Step 13

In Rails console run:

```
app.get '/'
```

Here we are simulating the browser GET request for the resource '/', which is your home page.

```
Started GET "/" for 127.0.0.1 at 2016-07-05 15:26:25 -0700
Processing by WelcomeController#index as HTML
  Rendering welcome/index.html.erb within layouts/application
  Rendered welcome/index.html.erb within layouts/application (1.4ms)
Completed 200 OK in 295ms (Views: 272.8ms | ActiveRecord: 0.0ms)
```

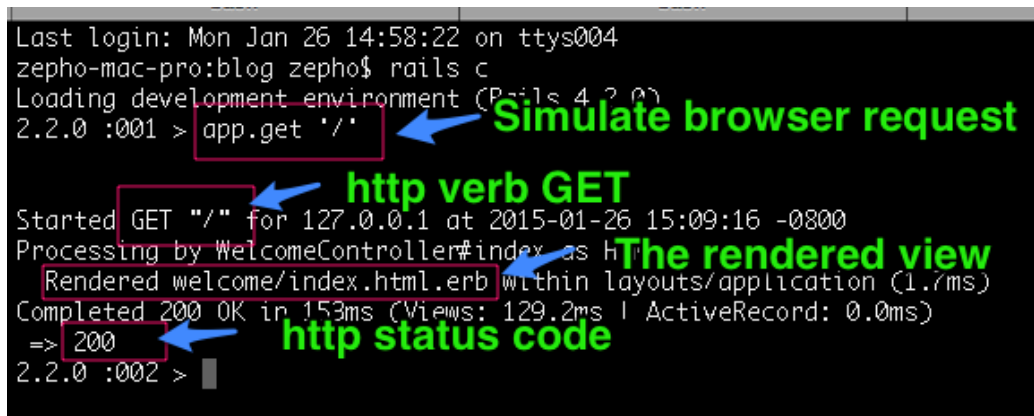
A screenshot of a terminal window showing a Rails console session. The terminal has a black background with white text. The session starts with a login message, then the user enters 'rails c' to enter the console. The prompt is '2.2.0 :001 >'. The user enters 'app.get '/''. A red box is drawn around this command, with a blue arrow pointing to it from the text 'Simulate browser request' in green. The output shows the request being processed, the view being rendered, and the status code 200. A red box is drawn around '200', with a blue arrow pointing to it from the text 'http status code' in green. Another red box is drawn around 'Rendered welcome/index.html.erb', with a blue arrow pointing to it from the text 'The rendered view' in green. The session ends with the prompt '2.2.0 :002 >'.

Figure 7: Simulating Browser GET Request

You can see the http status code is 200. You can also see which view was rendered for this request.

Exercise

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

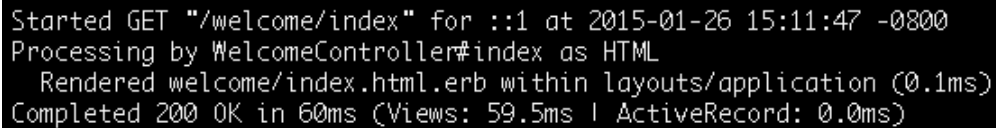
Before you go to the next page and read the answer, make an attempt to answer this question.

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 config/routes.rb file, you see :

```
get 'welcome/index'
```

This line in routes was added when you ran the rails generator to create the welcome controller. 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.



```
Started GET "/welcome/index" for ::1 at 2015-01-26 15:11:47 -0800
Processing by WelcomeController#index as HTML
  Rendered welcome/index.html.erb within layouts/application (0.1ms)
Completed 200 OK in 60ms (Views: 59.5ms | ActiveRecord: 0.0ms)
```

Figure 8: GET Request for Home Page

Step 14

Delete the get “welcome/index” line in the config/routes.rb file. Reload the page, by entering this URL in the browser : `http://localhost:3000/welcome/index`



Figure 9: Welcome Index

No route matches [GET] "/welcome/index"

You will now see the error page. Since we no longer need this route, we can ignore this error message because the home page will be accessed by typing just the domain name of your site like this: `www.mysite.com`

Summary

In this lesson we wrote a simple *Hello Rails* program. We saw how the router recognizes the browser request and how the view and controller work in Rails to handle it. We have seen just the View and Controller part of MVC framework. We will see how the model fits in the MVC framework in the next lesson.

CHAPTER 3

Model

Objective

- To learn the model part M of the MVC framework

Context

We are going to create an web application that will have articles to read, create, list, update and delete.

Steps

Step 1

In Rails, model is a persistent object that can also contain business logic. Model is the Object Relational Mapping (ORM) layer that uses ActiveRecord design pattern. Open config/routes.rb file and add :

```
resources :articles
```

Save the file. Your file should like this :

```
Rails.application.routes.draw do
  resources :articles

  root 'welcome#index'
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 a 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:

```
$ rails routes
```

```
$ rake routes
```

	Prefix	Verb	URI Pattern	Controller#Action
	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
	root	GET	/	welcome#index

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

```

zepho-mac-pro:rlog5 zepho$ rails routes
      Prefix Verb   URI Pattern
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)
root GET    /
Controller#Action
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
welcome#index

```

Figure 10: Installed Routes

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.

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

Based on the requirements you will choose either 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 ActiveRecord subclass by running

the following command:

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

bash	bash	rub
zepho-mac-pro:blog zepho\$ rails g model article title:string description:text		
invoke	active_record	
create	db/migrate/20150127005343_create_articles.rb	
create	app/models/article.rb	
invoke	test_unit	
create	test/models/article_test.rb	
create	test/fixtures/articles.yml	

Figure 11: Article Model

In this command the rails generator generates a model by the name of article. The ActiveRecord class name is in 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. The title will be displayed as a text field and the description will be displayed as text area in the view.

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. The class `CreateArticles` is a subclass of `ActiveRecord::Migration` class.

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 its data type.

In our case we are creating the `articles` table, `t.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 in the migration file.

Step 5

Go to the `blog` directory in the terminal and run :

```
$ rails db:migrate
```

You will see the output:

```
== 20160705224113 CreateArticles: migrating =====
-- create_table(:articles)
   -> 0.0017s
== 20160705224113 CreateArticles: migrated (0.0018s) ==
```

This will create the `articles` table.

bash	bash
<pre>zepho-mac-pro:blog zepho\$ rake db:migrate == 20150127005343 CreateArticles: migrating ===== -- create_table(:articles) -> 0.0015s == 20150127005343 CreateArticles: migrated (0.0016s) =====</pre>	

Figure 12: Create 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.

Step 7

In the database console run:

```
select count(*) from articles;
```

```
$rails db
SQLite version 3.7.7 2011-06-25 16:35:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select count(*) from articles;
0
```

You can see from the output there are no records in the database. If you want to exit the db console type `.quit`.

```
bash
zepho-mac-pro:blog zepho$ rails db
SQLite version 3.7.7 2011-06-25 16:35:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select count(*) from articles;
0
sqlite> █
```

Figure 13: Rails Db Console

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.

```
(0.1ms)  SELECT COUNT(*) FROM "articles"
=> 0
```

You can see that ActiveRecord generated the SQL query we used in Step 7. The count is 0. Let's create a new record in the articles table.

```

zepho-mac-pro:rlog5 zepho$ rails c
Loading development environment (Rails 5.0.0)
2.3.1 :001 > Article.count
(0.1ms) SELECT COUNT(*) FROM "articles"
=> 0

```

Figure 14: Rails Console

Step 10

Type :

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

```

2.2.0 :002 > Article.create(title: 'test', description: 'first row')
(0.2ms) begin transaction
SQL (0.5ms) INSERT INTO "articles" ("title", "description", "created_at", "updated_at")
VALUES (?, ?, ?, ?) [["title", "test"], ["description", "first row"], ["created_at",
"2015-01-27 01:10:35.836769"], ["updated_at", "2015-01-27 01:10:35.836769"]]
(35.2ms) commit transaction
=> #<Article id: 1, title: "test", description: "first row", created_at: "2015-01-27 01:
10:35", updated_at: "2015-01-27 01:10:35">
2.2.0 :003 >

```

Figure 15: Create a Record

```

(0.1ms) begin transaction
SQL (0.4ms) INSERT INTO "articles" ("title", "description", "created_at", "updated_at")
VALUES (?, ?, ?, ?)
(43.8ms) commit transaction
=> #<Article id: 1, title: "test", description: "first row", created_at: "2016-07-27 01:10:35", updated_at: "2016-07-27 01:10:35">

```

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

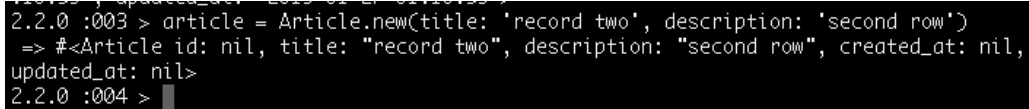
Exercise 1

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')
```

A screenshot of a terminal window with a black background and white text. It shows a Rails console session. The prompt is '2.2.0 :003 >'. The user enters the command 'article = Article.new(title: 'record two', description: 'second row')'. The prompt changes to '2.2.0 :004 >'. The output is '=> #<Article id: nil, title: "record two", description: "second row", created_at: nil, updated_at: nil>'.

```
2.2.0 :003 > article = Article.new(title: 'record two', description: 'second row')
=> #<Article id: nil, title: "record two", description: "second row", created_at: nil,
updated_at: nil>
2.2.0 :004 > █
```

Figure 16: Article Instance

You will see this output:

```
=> #<Article id: nil, title: "record two", description: "second row", created_at:
```

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
<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 > Article.count (0.6ms) SELECT COUNT(*) FROM "articles" => 1 2.0.0p247 :009 > _</pre>		

Figure 17: Article Count

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

```
article.save
```

```
2.2.0 :005 > article.save
(0.2ms) begin transaction
SQL (0.4ms) INSERT INTO "articles" ("title", "description", "created_at", "updated_at") VALUES (?, ?, ?, ?) [["title", "record two"], ["description", "second row"], ["created_at", "2015-01-27 01:14:20.102718"], ["updated_at", "2015-01-27 01:14:20.102718"]]
(39.1ms) commit transaction
=> true
```

Figure 18: Saving a Record

```
(0.1ms) begin transaction
SQL (0.5ms) INSERT INTO "articles" ("title", "description", "created_at", "updated_at") VALUES (?, ?, ?, ?) [["title", "record two"], ["description", "second row"], ["created_at", "2015-01-27 01:14:20.102718"], ["updated_at", "2015-01-27 01:14:20.102718"]]
(50.2ms) commit transaction
=> true
```

Now query the articles table to get the number of records. You should now have two records in the database.

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 display all the records in articles table on the browser. We will also see how the different parts of the MVC interact to create database driven dynamic web application.

CHAPTER 4

Model View Controller

Objectives

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

Context

Router knows which controller can handle the incoming request. Controller is like a traffic cop who controls the flow of traffic on busy streets. Controller has the knowledge of which model can get the job done, so it delegates the work to the appropriate model object. Controller also knows which view to display to the user after the incoming request has been processed.

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 `app/views/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 is 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's use the output of rake routes to find the name of the view_helper we can use.

Prefix	Verb	URI Pattern	Controller#Action
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
root	GET	/	welcome#index

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

```

zepho-mac-pro:rlog5 zepho$ rails routes
      Prefix Verb   URI Pattern
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)
root GET    /

```

Figure 19: 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.

What do you see in the home page?



Figure 20: My Blog

You will see the hyper link in the home page.

Step 4

Right click and do 'View Page Source' in Chrome or 'Show Page Source' in Safari.

You will see the hyperlink which is a relative url.

```
html <a href="/articles">My Blog</a>
```



```
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="/articles">My Blog</a>
20
21  </body>
22 </html>
23
```

Figure 21: Page Source for Relative URL

Step 5

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

Reload the page. View the page source again.

```
<a href="http://localhost:3000/articles">My Blog</a>
```

You will now see the absolute 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 22: Page Source for Absolute URL

Step 6

Click on the ‘My Blog’ link.

You will see the above error page with ‘uninitialized constant ArticlesController’ error.

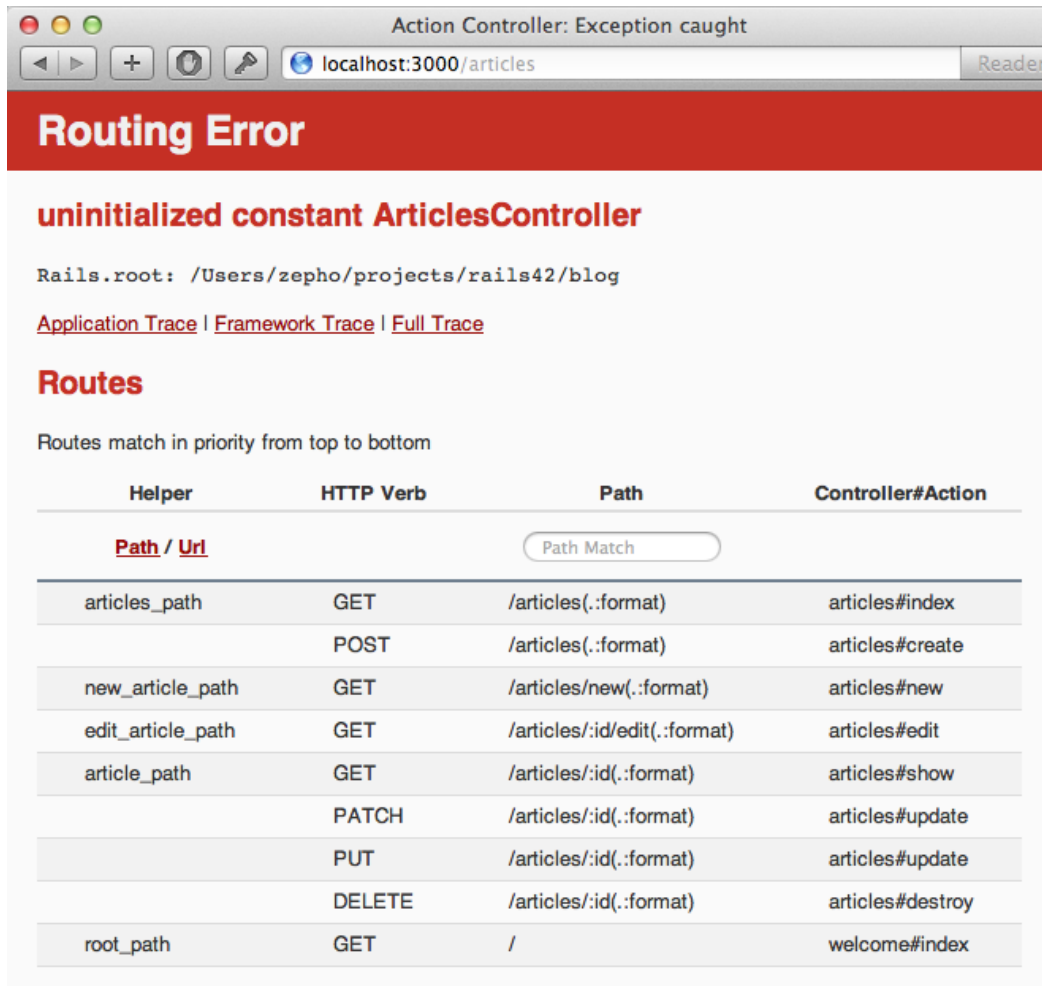
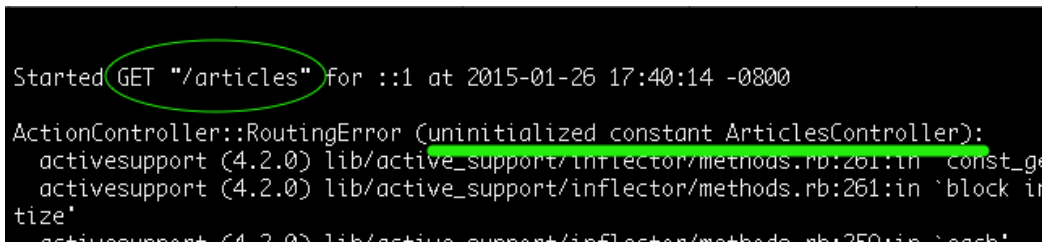


Figure 23: Routing Error

Step 7

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

A screenshot of a terminal window showing the Rails server log. The first line is "Started GET \"/articles\" for ::1 at 2015-01-26 17:40:14 -0800", where the GET method is circled in green. The second line is "ActionController::RoutingError (uninitialized constant ApplicationController):", where the error message is highlighted in green. The following lines show the stack trace from ActiveSupport (4.2.0) lib/active_support/inflector/methods.rb:261:in `const_get' and ActiveSupport (4.2.0) lib/active_support/inflector/methods.rb:261:in `block in size'.


```
Started GET "/articles" for ::1 at 2015-01-26 17:40:14 -0800
ActionController::RoutingError (uninitialized constant ApplicationController):
  ActiveSupport (4.2.0) lib/active_support/inflector/methods.rb:261:in `const_get'
  ActiveSupport (4.2.0) lib/active_support/inflector/methods.rb:261:in `block in size'
  ActiveSupport (4.2.0) lib/active_support/inflector/methods.rb:259:in `each'
```

Figure 24: Articles Http Request

```
Started GET "/articles" for ::1 at 2016-07-05 16:12:20 -0700
```

```
ActionController::RoutingError (uninitialized constant ApplicationController):
```

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.



#	Method	Status	Url
2	GET	200	http://localhost:3000/favicon.ico
1	GET	200	http://localhost:3000/articles

Figure 25: 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 26: Live HTTP Headers Showing Client Server Interaction

Here you see the client-server interaction details. As you see in the above figure, 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 27: Live HTTP Headers Gives Ton of Information

Step 8

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

```
zepho-mac-pro:rlog5 zepho$ 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
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/articles.coffee
  invoke  scss
  create  app/assets/stylesheets/articles.scss
```

Figure 28: Generate Controller

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

Step 9

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



Figure 29: 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` in the `app/controllers` directory 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 by Rails. 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 code editor and add the following code:

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

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

  <%= article.title %>

  <%= article.description %>
<br/>
<% 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 in the browser using `<%= %>` tags.

If you make a mistake and use `<%= %>` tags instead of Ruby scriptlet tag in `app/views/index.html.erb` like this:

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

You will see the objects in the array displayed on the browser.



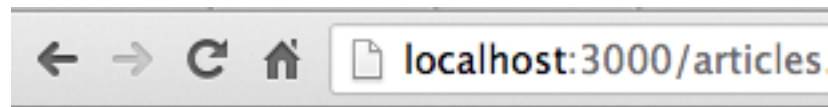
Figure 30: Using the Wrong Tags

Articles are displayed as objects inside an array.

If you use the Ruby scriptlet tag :

Title : <% article.title %>

instead of the tags used to evaluate expressions and display to the browser then you will not see it in the browser.



Listing Articles

Title :

Description : duplication

[Edit](#) [Show](#) [Delete](#)

[New Article](#)

Figure 31: No Title Value in Browser

Step 12

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



Figure 32: List of Articles

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

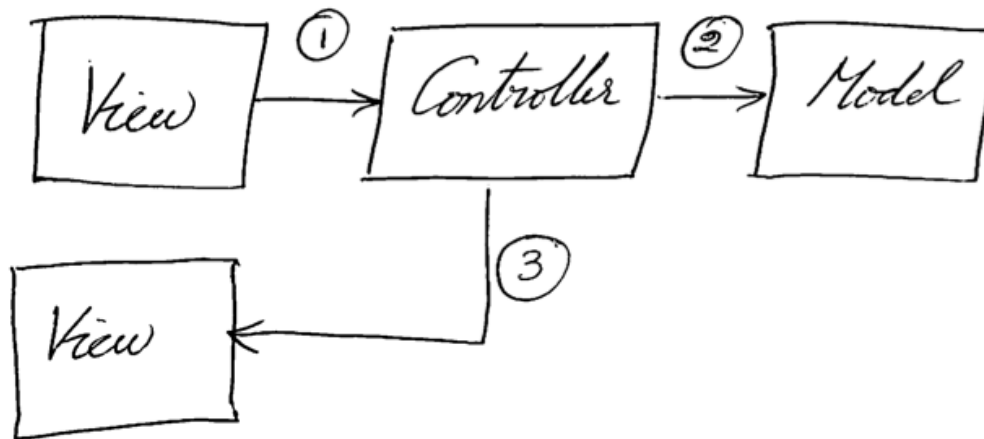


Figure 33: Model View Controller

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.

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?

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 interact 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 the 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.

CHAPTER 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 rails routes output:

bash	bash	ruby	...	bash
zepho-mac-pro:blog zepho\$ rake routes				
	Prefix Verb	URI Pattern		Controller#Action
articles_index	GET	/articles/index(:format)		articles#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
root	GET	/		welcome#index

Figure 34: 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.



Figure 35: 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'.

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

  test
  first row
<br/>
  record two
  second row
<br/>
<a href="/articles/new">New Article</a>
.. .
```

Figure 36: View Page Source

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

```
<a href='/articles/new'>New Article</a>
```

Step 5

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

```
Started GET "/articles/new" for ::1 at 2015-01-26 18:10:03 -0800
```

Figure 37: HTTP Verb Get

```
Started GET '/articles/new' for ::1 at 2016-07-05 16:33:59 -0700
```

```
AbstractController::ActionNotFound (The action 'new' could not be found for ArticlesController)
```

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



Figure 38: Action New Not Found

You will see the unknown action 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

Go back to <http://localhost:3000/articles> page. Click on the 'New Article' link, in the log, you will see:

```
Started GET '/articles/new' for ::1 at 2016-07-05 16:34:54 -0700  
Processing by ArticlesController#new as HTML  
Completed 406 Not Acceptable in 55ms (ActiveRecord: 0.0ms)
```

```
ActionController::UnknownFormat (ArticlesController#new is missing a template for
```

```
request.formats: ['text/html']  
request.variant: []
```

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`. Since this view is missing we don't see the page with the form to fill out the article fields.

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

Click on the 'New Article' link on the 'Listing Articles' page.

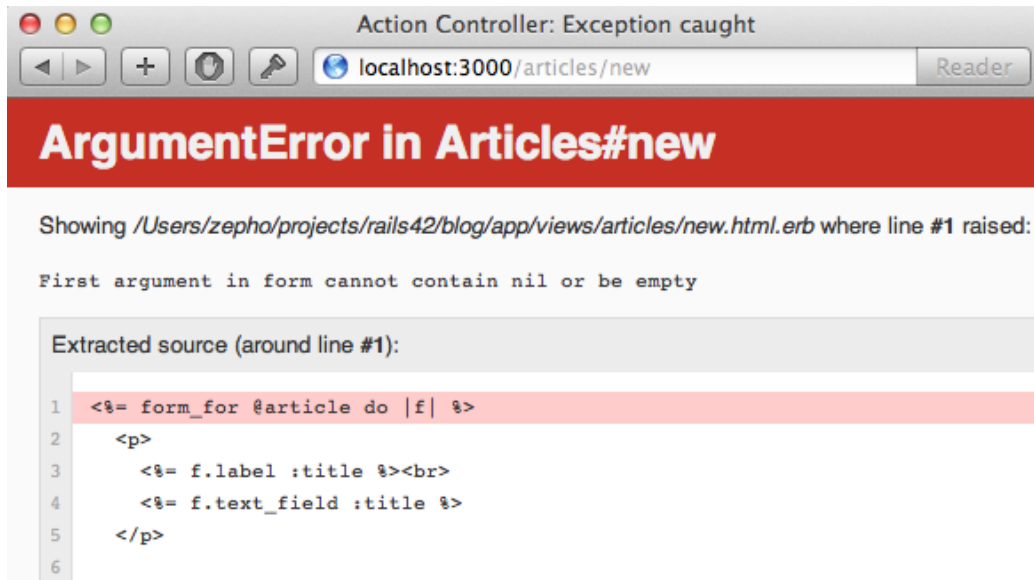


Figure 39: Argument Error

ActionView::Template::Error (First argument in form cannot contain nil or be empty)

```
1: <%= form_for @article do |f| %>
2:   <p>
3:     <%= f.label :title %><br>
4:     <%= f.text_field :title %>
```

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 title 'Blog'. The address bar displays 'localhost:3000/articles/new'. The main content area contains a form with two input fields: 'Title' and 'Description'. The 'Title' field is a single-line text input, and the 'Description' field is a multi-line text area. Below these fields is a button labeled 'Create 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>
```

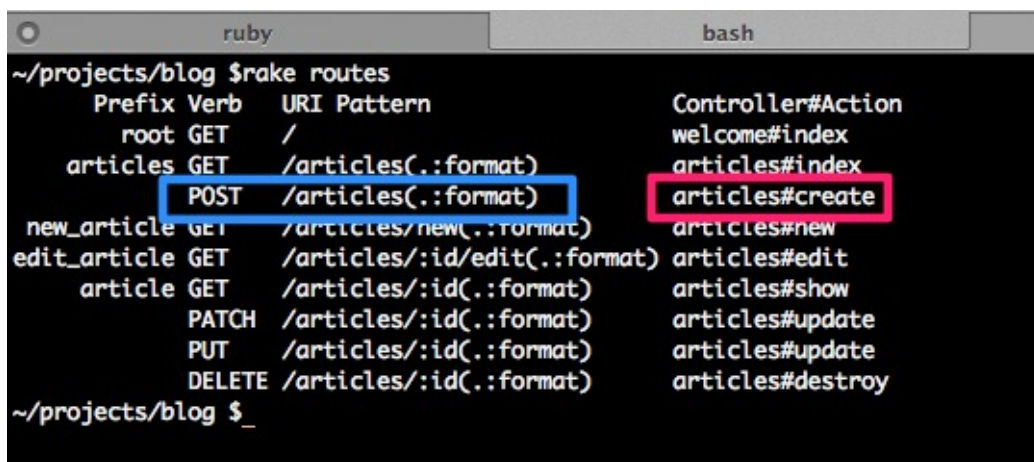
Figure 41: New Article Page Source

```
<form class='new_article' id='new_article' action='/articles' accept-charset='UTF-
```

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 rails 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
      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 $
```

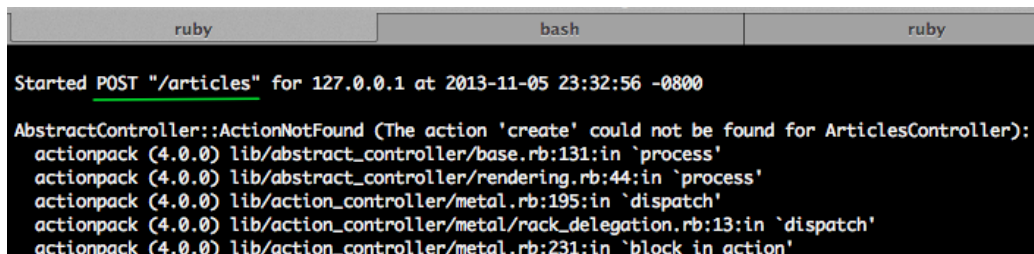
Figure 42: Identifying Resource Endpoint

```
POST    /articles(.:format)      articles#create
```

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

```
Started POST '/articles' for ::1 at 2016-07-05 16:40:29 -0700
```

```
AbstractController::ActionNotFound (The action 'create' could not be found for Art
```

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 the create action in the articles controller.

Step 15

Define the create method in the articles controller as follows:

```
“ruby def create
```

```
end “ ## Step 16 ##
```

Fill out the form and click ‘Create Article’.

ruby	bash	ruby
<pre>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"=>"6gubQ3YqRqy0RqwhYYyMiy+NEDkNmGbYcj0PXQg8TBg=", "article"={"title"=>"test", "description"=>"tester1"}, "commit"=>"Create Article"} Completed 500 Internal Server Error in 2ms</pre>		

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 database columns and its values.

```
Started POST '/articles' for ::1 at 2016-07-05 16:41:31 -0700
Processing by ArticlesController#create as HTML
Parameters: {'authenticity_token'=>'1ILcWt9sezSepVcNdV', 'article'=>{'title'=>'s
No template found for ArticlesController#create, rendering head :no_content
Completed 204 No Content in 56ms (ActiveRecord: 0.0ms)
```

You will see No template found in the log file.

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 that 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
```

In Figure 50, you see that the hash key `article` is a string, but I am using the symbol `:article` in the `create` method. How does this work?

ruby	...	bash
<pre>2.0.0p247 :004 > x = ActiveSupport::HashWithIndifferentAccess.new => {} 2.0.0p247 :005 > x['score'] = 10 => 10 2.0.0p247 :006 > x[:score] => 10 2.0.0p247 :007 > y = {} => {} 2.0.0p247 :008 > y['score'] = 5 => 5 2.0.0p247 :009 > y[:score] => nil 2.0.0p247 :010 ></pre>		

Figure 46: HashWithIndifferentAccess

As you can see from the rails console, `params` hash is not a regular Ruby hash, it is a special hash called `HashWithIndifferentAccess`. It allows you to set the value of the hash with either a symbol or string and retrieve the value using either string or symbol.

Step 18

Fill out the form and submit again.

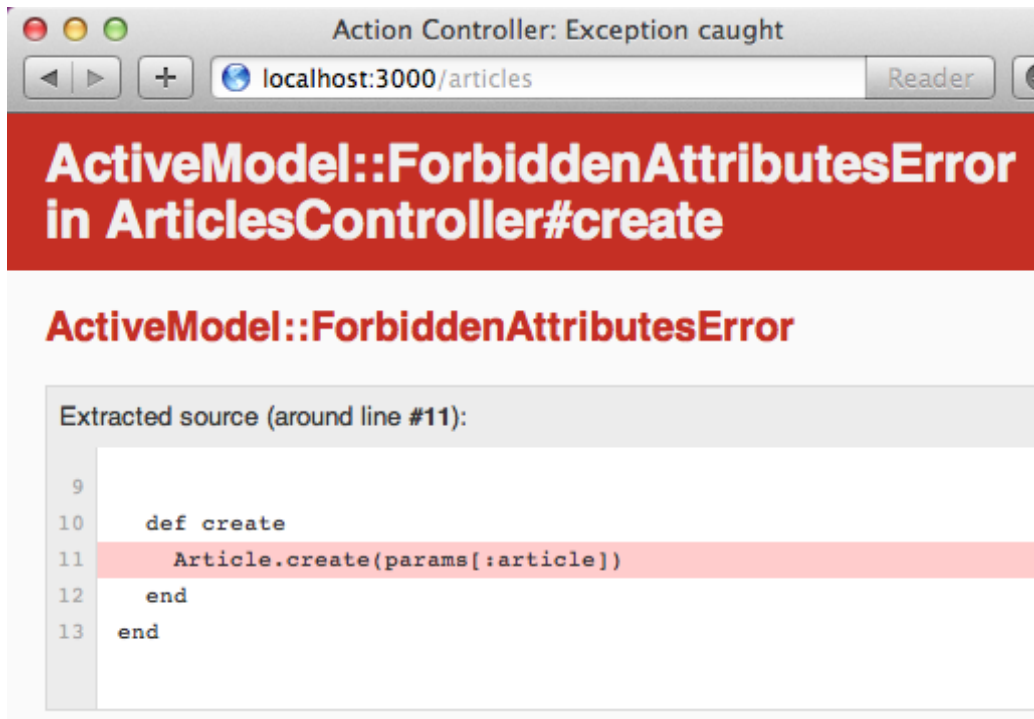


Figure 47: Forbidden Attributes Error

Now we get a `ActiveModel::ForbiddenAttributesError` 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 no template error but you will now see that the user submitted data is saved in the database.

```
Started POST '/articles' for ::1 at 2016-07-05 16:44:10 -0700
Processing by ArticlesController#create as HTML
  Parameters: {'authenticity_token'=>'b1XvtCYVw5P0GYF028L', 'article'=>{'title'=>'
  (0.1ms) begin transaction
  SQL (0.5ms) INSERT INTO 'articles' ('title', 'description', 'created_at', 'update
  (48.8ms) commit transaction
No template found for ArticlesController#create, rendering head :no_content
Completed 204 No Content in 122ms (ActiveRecord: 50.3ms)
```



```
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

Let's go to the rails console and check the record that we just created.

```
$ rails c
Loading development environment (Rails 5.0.0)
>> Article.last
Article Load (0.2ms) SELECT 'articles'.* FROM 'articles' ORDER BY 'articles'.
=> #<Article id: 3, title: 'Basics of Abstraction', description: 'testing', crea
```

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

Step 21

Let's now address the no template 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 rails routes command, we know the resource end point, to find the URL helper we look at the Prefix column.

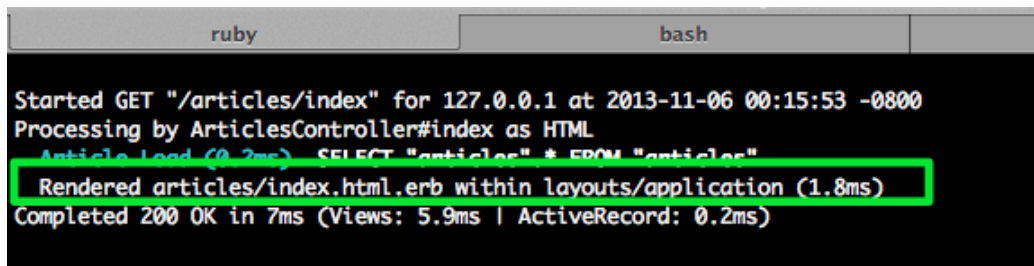
Step 22

Fill out the form and submit again.



Figure 49: Displaying All Articles

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



```
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.

Exercise

Open the routes.rb file and delete the line:

```
get 'articles/index'
```

Change the helper name for `redirect_to` method in create action by looking at the

```
rails routes
```

output.

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 action works. In the next lesson we will see how edit and update action works to make changes to an existing record in the database.

CHAPTER 6

Update Article

Objective

- Learn how to update an existing record in the database

Steps

Step 1

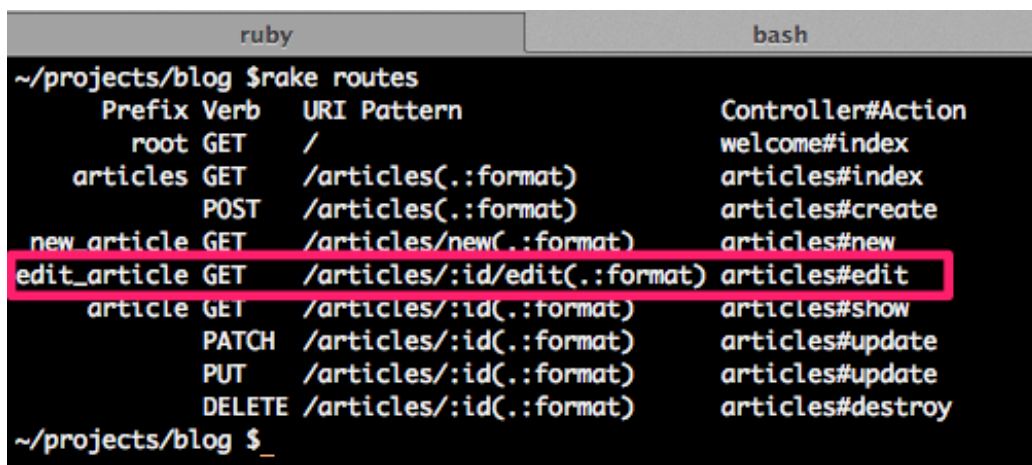
Let's add 'Edit' link to each record that is displayed in the index page. Open the `app/views/articles/index.html.erb` and add the edit link:

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

What should be the url helper to use in the second parameter to the `link_to` method?

Step 2

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 rails routes output, the Prefix column gives us the url helper to 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 51: Edit Article URL Helper

So we now have:

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

Step 3

Go to Rails console and type:

```
app.edit_article_path
```

```
$rails c
```

```
Loading development environment (Rails 5.0.0.beta1)
```

```
> app.edit_article_path
```

```
ActionController::UrlGenerationError: No route matches {:action=>"edit", :controller=>"articles"}
```

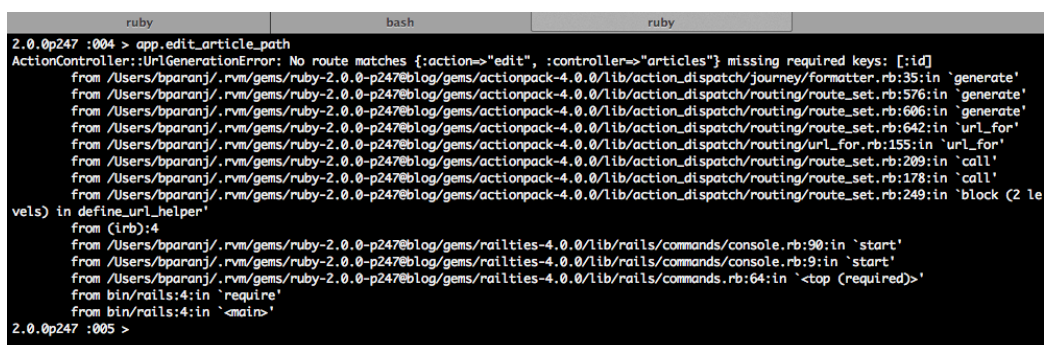
A screenshot of a terminal window showing a Rails console session. The prompt is '2.0.0p247 :004 >'. The user has entered 'app.edit_article_path'. The console displays an 'ActionController::UrlGenerationError' with the message 'No route matches {:action=>"edit", :controller=>"articles"} missing required keys: [:id]'. Below this, a stack trace is shown, listing various files and line numbers, including 'journey/formatter.rb:35', 'route_set.rb:576', 'route_set.rb:606', 'route_set.rb:642', 'url_for.rb:155', 'route_set.rb:209', 'route_set.rb:178', and 'route_set.rb:249'. The error occurs within the 'define_url_helper' method. The terminal window has tabs labeled 'ruby', 'bash', and 'ruby' at the top.

Figure 52: Edit Article URL Helper Error

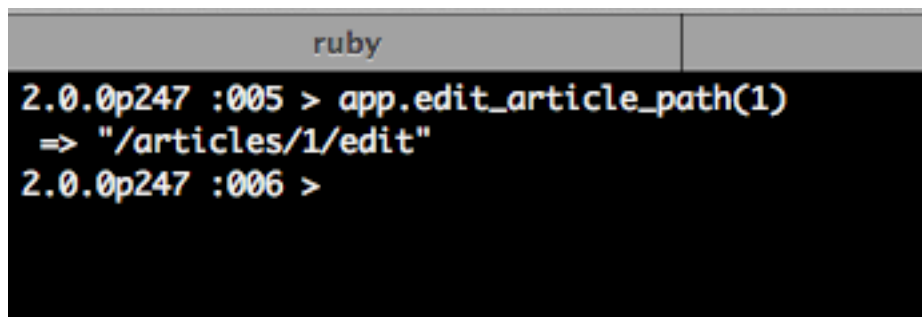
Rails does not recognize `edit_article_path` helper method.

Step 4

Examine the output of rake routes command. 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 an 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. Modify the link_to method as follows:

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

A screenshot of a terminal window with a dark background. The title bar at the top is light gray and contains the word "ruby". The terminal shows a prompt "2.0.0p247 :005 >" followed by the command "app.edit_article_path(1)". The output is "=> \"/articles/1/edit\"", and the prompt continues with "2.0.0p247 :006 >".

```
ruby
2.0.0p247 :005 > app.edit_article_path(1)
=> "/articles/1/edit"
2.0.0p247 :006 >
```

Figure 53: Edit Article URL Helper

Rails recognizes edit_article_path when the primary key :id value is passed in as the argument.

Step 5

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 6

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



Figure 54: Edit Article Link

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

Step 7

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 55: Edit Article Page Source

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

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

  test
  first row
  <a href="/articles/1/edit">Edit</a>
```

```
<br/>

record two
second row
<a href="/articles/2/edit">Edit</a>

<a href="/articles/new">New Article</a>

</body>
```

Step 8

Click on the 'Edit' link.



Figure 56: Unknown Action Edit

You will see The action 'edit' could not be found for ArticlesController error page.

Step 9

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

```
def edit  
  
end
```

Step 10

Click on the ‘Edit’ link. You now get the following error in the rails server log.

```
Started GET "/articles/1/edit" for ::1 at 2016-07-05 17:33:25 -0700
Processing by ArticlesController#edit as HTML
  Parameters: {"id"=>"1"}
Completed 406 Not Acceptable in 56ms (ActiveRecord: 0.0ms)
```

ActionController::UnknownFormat (ArticlesController#edit is missing a template for

```
request.formats: ["text/html"]
request.variant: []
```

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 11

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

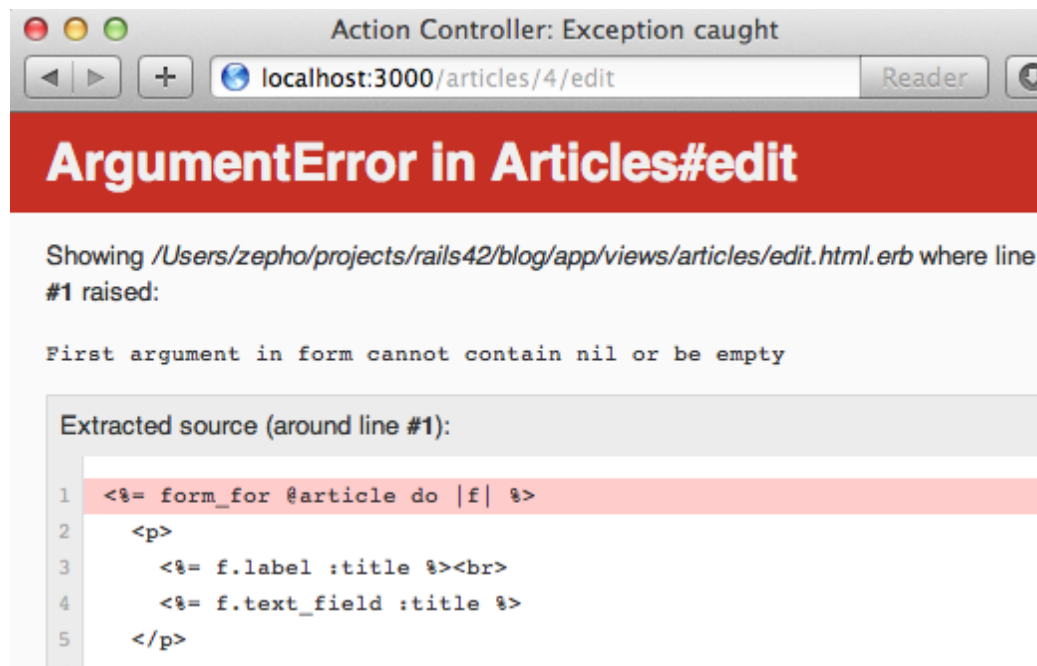


Figure 57: Argument Error in Articles Edit

```
ActionView::Template::Error (First argument in form cannot contain nil or be empty)
1: <%= form_for @article do |f| %>
2:   <p>
3:     <%= f.label :title %><br>
4:     <%= f.text_field :title %>
```

```
app/views/articles/edit.html.erb:1:in ‘_app_views_articles_edit_html_erb__29158’
```

We are getting this error because the first argument to `form_for` helper method cannot be `nil` or empty. In our case, it is `nil`. You can verify this by typing:

```
@article
```

in the Webconsole on the browser.

Step 12

Look at the server log:

bash	bash	ruby	bash
<pre>Started GET "/articles/4/edit" for ::1 at 2015-01-26 20:13:14 -0800 Processing by ArticlesController#edit as HTML Parameters: {"id"=>"4"} Rendered articles/edit.html.erb within layouts/application (5.2ms) Completed 500 Internal Server Error in 15ms</pre>			

Figure 58: Edit Article Server Log

```
Started GET "/articles/1/edit" for ::1 at 2016-07-05 17:35:12 -0700
Processing by ArticlesController#edit as HTML
  Parameters: {"id"=>"1"}
  Rendering articles/edit.html.erb within layouts/application
  Rendered articles/edit.html.erb within layouts/application (2.4ms)
Completed 500 Internal Server Error in 8ms (ActiveRecord: 0.0ms)
```

You can see that the primary key of the selected article id and its value.

bash	bash	ruby	bash
<pre>Started GET "/articles/4/edit" for ::1 at 2015-01-26 20:13:14 -0800 Processing by ArticlesController#edit as HTML Parameters: {"id"=>"4"} Rendered articles/edit.html.erb within layouts/application (5.2ms) Completed 500 Internal Server Error in 15ms</pre>			

Figure 59: Params Hash Populated by Rails

Rails automatically populates params hash and makes it available to the controllers. In this case, the params hash will contain id as the key and 1 as its value.

Step 13

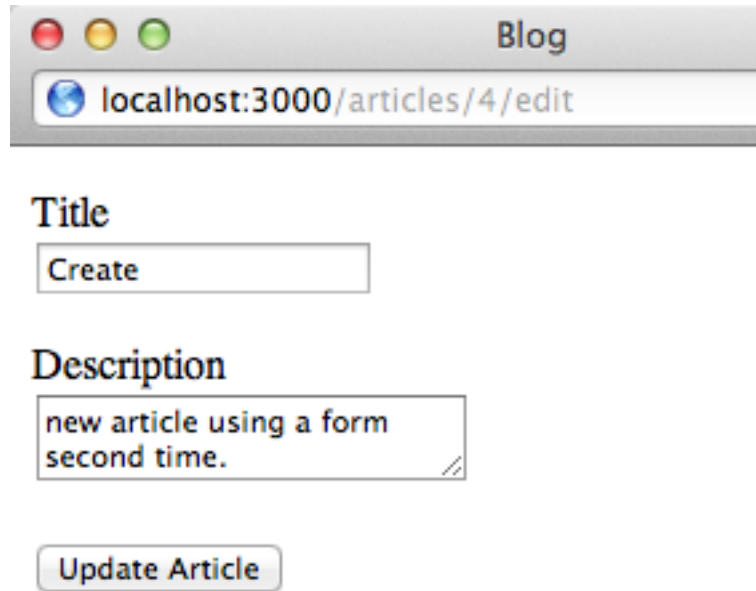
In the edit action we need to load the selected record from the database so that we can display it with the existing values for its columns. You already know that Rails populates params hash with the values submitted in the GET request for resource `‘/articles/1/edit’`. We can now define the edit method 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 14

Click on the 'Edit' link.



The screenshot shows a web browser window with the title 'Blog'. The address bar displays 'localhost:3000/articles/4/edit'. Below the address bar, there is a form with two main sections: 'Title' and 'Description'. The 'Title' section has a text input field containing the word 'Create'. The 'Description' section has a text area containing the text 'new article using a form second time.'. At the bottom of the form, there is a button labeled 'Update Article'.

Figure 60: Edit Article Form

You will now see the form with values populated.

Step 15

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
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 61: Edit Article Source

```
<form class="edit_article" id="edit_article_1" action="/articles/1" accept-charset="UTF-8">
```

We see that the URI pattern is ‘/articles/1’ and the http verb is POST. If you look at the output of rails 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.

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.

```
<input type="hidden" name="_method" value="patch" />
```

If you look at the output of rails routes, for the combination of `PATCH` and ‘/articles/1’ you will see 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
      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 62: HTTP Verb POST

```
zepho-mac-pro:rlog5 zepho$ rails routes
      Prefix Verb   URI Pattern                      Controller#Action
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
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
      root   GET    /                                welcome#index
```

Figure 63: HTTP Verb PATCH

```
PATCH /articles/:id(.:format) articles#update
```

Rails 5 uses PATCH instead of PUT that it used in previous versions. This is because PUT is an idempotent operation.

An idempotent HTTP method is a HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. - The RESTful Cookbook by Joshua Thijsen

Step 16

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. Add the following update method to articles controller.

```
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 17

Go to articles index page by going to <http://localhost:3000/articles>. Click on the 'Edit' link. In the edit form, you can change the value of either the title or description and click 'Update Article' button.

Step 18

We can see in the console:

```
Started PATCH "/articles/1" for ::1 at 2016-07-05 17:44:22 -0700
Processing by ArticlesController#update as HTML
```

The combination of PATCH http verb and `/articles/1` routed the request to the update action of the articles controller.

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 in the console, the template is missing but the record has been successfully updated.

```
SQL (0.3ms)  UPDATE "articles" SET "title" = ?, "updated_at" = ? WHERE "articles".
(255.5ms)  commit transaction
No template found for ArticlesController#update, rendering head :no_content
Completed 204 No Content in 325ms (ActiveRecord: 256.9ms)
```

Step 19

Let's address the no template found 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 20

Edit the article and click ‘Update Article’. You should see that it now updates the article and redirects the user to the articles index page.

Step 21

An annoying thing about Rails 5 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. Because we already have resources :articles declaration in the routes.rb, let’s delete the following line:

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

in the config/routes.rb file. Now we longer have the articles_index_path helper, let’s 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 by displaying a form for an existing article and saving the new values in the database. In the next lesson we will see how to display a given article.

CHAPTER 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.

Which url helper should we use?

You can view the output of rails 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.

```
zepho-mac-pro:rlog5 zepho$ rails routes
      Prefix Verb   URI Pattern
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)
root GET    /
Controller#Action
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
welcome#index
```

Figure 64: URL Helper For Show

So, we now have :

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

Step 2

Go to Rails console and type:

```
> app.article_path
```

```
ActionController::UrlGenerationError: No route matches {:action=>"show", :controller=>"articles"}
```

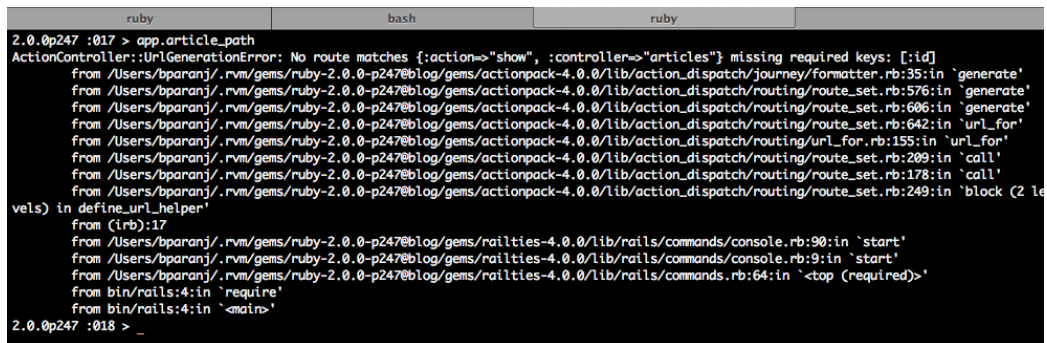
A screenshot of a terminal window showing a Rails console session. The prompt is '2.0.0p247 :017 >'. The user has entered 'app.article_path'. The console has outputted an error: 'ActionController::UrlGenerationError: No route matches {:action=>"show", :controller=>"articles"} missing required keys: [:id]'. Below the error message, there is a stack trace with several lines of code, including 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/journey/formatter.rb:35:in `generate\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/route_set.rb:576:in `generate\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/route_set.rb:606:in `generate\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/route_set.rb:642:in `url_for\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/url_for.rb:155:in `url_for\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/route_set.rb:209:in `call\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/route_set.rb:178:in `call\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_dispatch/routing/route_set.rb:249:in `block (2 levels) in define_url_helper\'', 'from (irb):17', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/railties-4.0.0/lib/rails/commands/console.rb:90:in `start\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/railties-4.0.0/lib/rails/commands/console.rb:9:in `start\'', 'from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/railties-4.0.0/lib/rails/commands/console.rb:64:in `<top (required)>\'', 'from bin/rails:4:in `require\'', 'from bin/rails:4:in `main\''. The prompt at the bottom is '2.0.0p247 :018 > _'.

Figure 65: Article Path Error

Rails does not recognize the `article_path`.

Step 3

Look at the output of rails routes command. You can see in the URI pattern column the :id variable for primary key.

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 66: Show Article Path Primary Key

```
article GET    /articles/:id(.:format)    articles#show
```

So we need to pass the id as the parameter as shown below:

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

ruby	bash	ruby
2.0.0p247 :019 > app.article_path(Article.first.id)		
Article Load (0.2ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id" ASC LIMIT 1		
=> "/articles/5"		
2.0.0p247 :020 > _		

Figure 67: Show Article Path

Rails recognizes article path when an id is passed in as the parameter to the url helper method.

```
<a href="/articles/1">Show</a>
```

You can see the generated string is the same as the URI pattern in the output of the rails routes command.

ruby	bash	ruby
<pre>2.0.0p247 :018 > app.article_path(Article.first) Article Load (0.3ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id" ASC LIMIT 1 => "/articles/5" 2.0.0p247 :019 ></pre>		

Figure 68: Show Article Path

We can simplify it even further by letting Rails call the id method for us by just passing the article object.

Step 4

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

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

Step 5

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

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

Let's now see how Rails makes this magic happen.

ruby	bash	ruby
<pre>2.0.0p247 :013 > article = Article.first Article Load (1.1ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id" ASC LIMIT 1 => #<Article id: 5, title: "not", description: "duplication", created_at: "2013-11-09 19:17:12", updated_at: "2013-11-09 19:17:23"> 2.0.0p247 :014 > _</pre>		

Figure 69: Loading First Article from Database

```
> a = Article.first
Article Load (0.3ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id" ASC LIMIT 1
=> #<Article id: 1, title: "test", description: "first row update", created_at: "2013-11-09 19:17:12", updated_at: "2013-11-09 19:17:23">
```

Retrieving first article from database in Rails console.

ruby	ba
<pre>2.0.0p247 :014 > app.polymorphic_path(article) => "/articles/5" 2.0.0p247 :015 ></pre>	

Figure 70: Show Article Path

```
> app.article_path(a.id)
=> "/articles/1"
> app.article_path(a)
=> "/articles/1"
```

Experimenting in Rails console to check the generated URI for a given article resource.

Rails internally uses the `polymorphic_path` method that takes an argument to generate the url.

Step 6

Add the link to display the article. 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_path(article) %>

  <br/>

<% end %>
<br/>
<%= link_to 'New Article', new_article_path %>
```

Step 7

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



Figure 71: Show Link

You will see the show link.

Step 8

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.

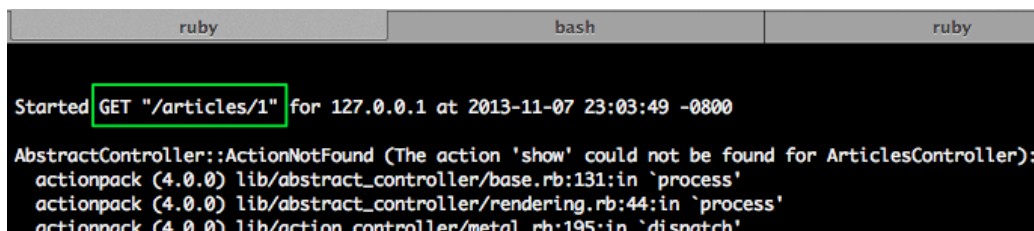


```
19 <h1>Listing Articles</h1>
20
21
22     test :
23
24     first row updated 2
25
26     <a href="/articles/1/edit">Edit</a>
27     <a href="/articles/1">Show</a>
28
29     <br/>
```

Figure 72: 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 ::1 at 2016-07-05 19:32:52 -0700
```



```
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 73: Http GET Request

Server log is another friend.

Step 9

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

```
AbstractController::ActionNotFound (The action 'show' could not be found for ArticleController):
```

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.

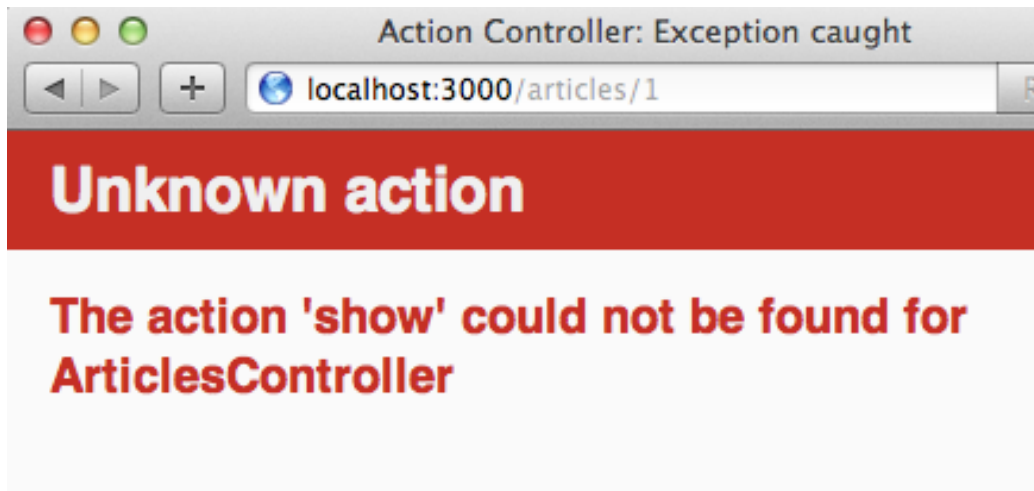


Figure 74: Unknown Action Show

Step 10

If you click the ‘Show’ link, you will get the error:

```
ActionController::UnknownFormat (ArticlesController#show is missing a template for
```

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.

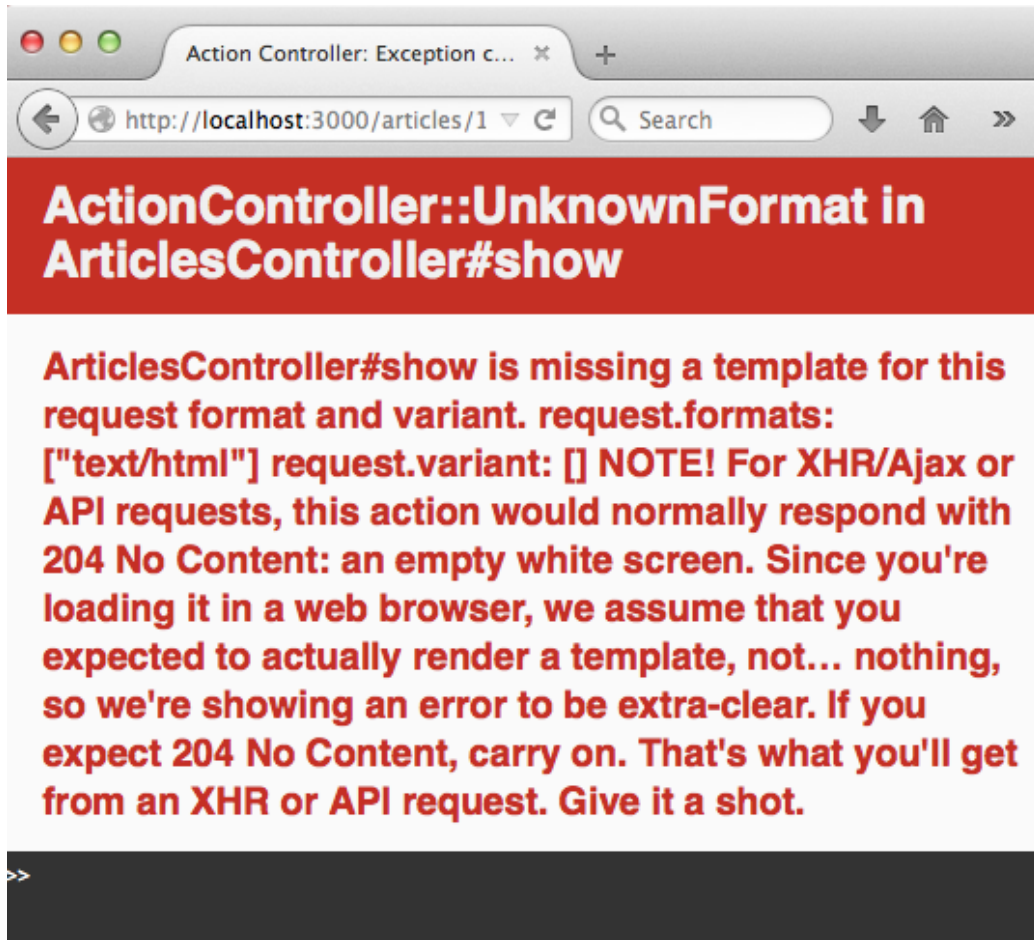


Figure 75: Unknown Format Error

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.

CHAPTER 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.

```

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

Figure 76: URL Helper For Delete

	Prefix	Verb	URI Pattern	Cont
	article_comments	GET	/articles/:article_id/comments(.:format)	comm
		POST	/articles/:article_id/comments(.:format)	comm
	new_article_comment	GET	/articles/:article_id/comments/new(.:format)	comm

edit_article_comment	GET	/articles/:article_id/comments/:id/edit(..format)	comm
article_comment	GET	/articles/:article_id/comments/:id(..format)	comm
	PATCH	/articles/:article_id/comments/:id(..format)	comm
	PUT	/articles/:article_id/comments/:id(..format)	comm
	DELETE	/articles/:article_id/comments/:id(..format)	comm
articles	GET	/articles(..format)	arti
	POST	/articles(..format)	arti
new_article	GET	/articles/new(..format)	arti
edit_article	GET	/articles/:id/edit(..format)	arti
article	GET	/articles/:id(..format)	arti
	PATCH	/articles/:id(..format)	arti
	PUT	/articles/:id(..format)	arti
	DELETE	/articles/:id(..format)	arti

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 %>

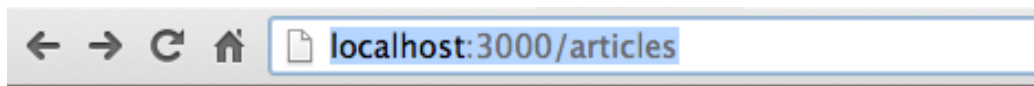
  <%= link_to 'Edit', edit_article_path(article) %>
  <%= link_to 'Show', article %>
  <%= link_to 'Delete',
    article_path(article),
    method: :delete,
    data: { confirm: 'Are you sure?' } %>

  <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 77: Delete Link

The delete link in the browser.

Step 3

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

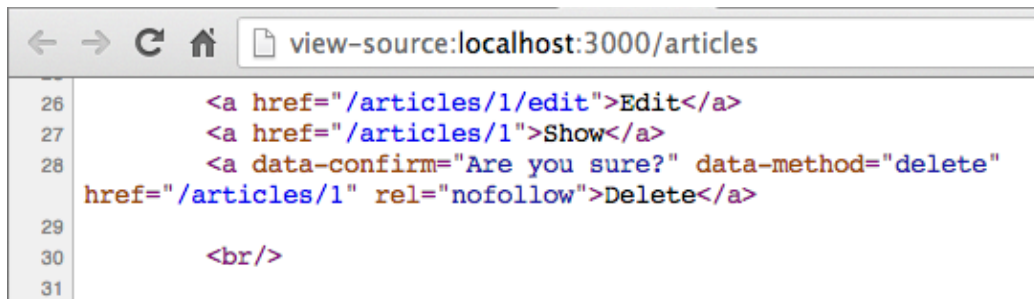


Figure 78: Delete Link Page Source

```
<a data-confirm="Are you sure?" rel="nofollow" data-method="delete" href="/article
```

You see the html5 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 `/assets/jquery_ujs.self-xyz.js` link.

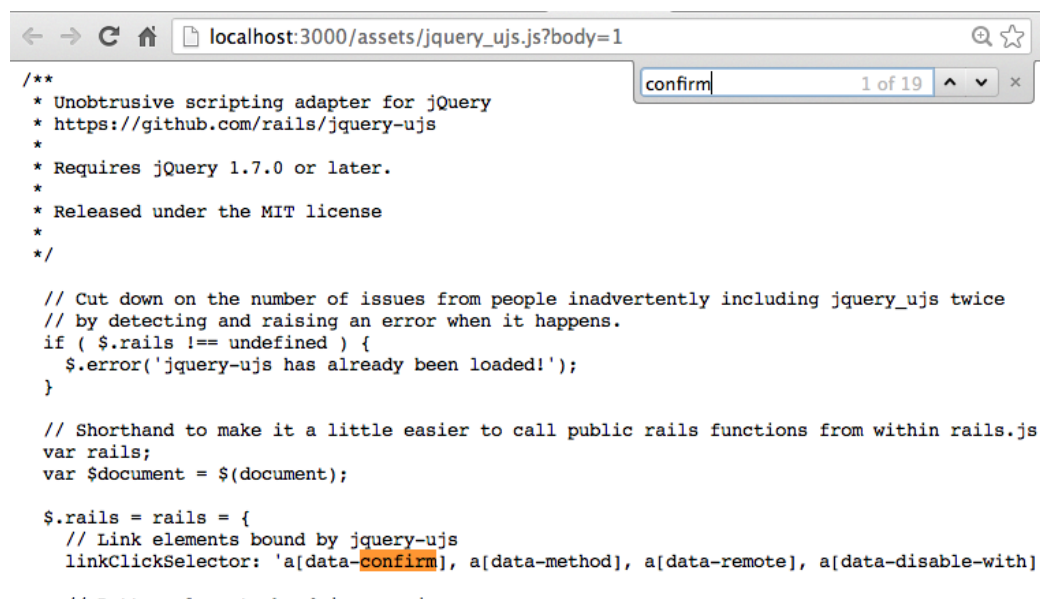


Figure 79: Data Confirm Link Element

Search for ‘confirm’. The first occurrence shows you the link element bound by jquery-ujs.

```
// Link elements bound by jquery-ujs
  linkClickSelector: 'a[data-confirm], a[data-method], a[data-remote]:not([disab
```

UJS stands for Unobtrusive Javascript. It is unobtrusive because you don’t see any javascript code in the html page.

```
javascript // Default confirm dialog, may be overridden with
custom confirm dialog in $.rails.confirm confirm: function(message)
{ return confirm(message); }, If you scroll down you the see default
confirm dialog as shown in the above code snippet.
```

You can search for ‘data-method’.

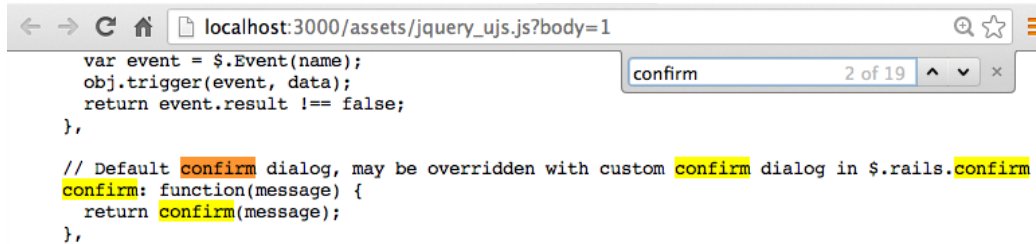


Figure 80: Data Confirm Popup

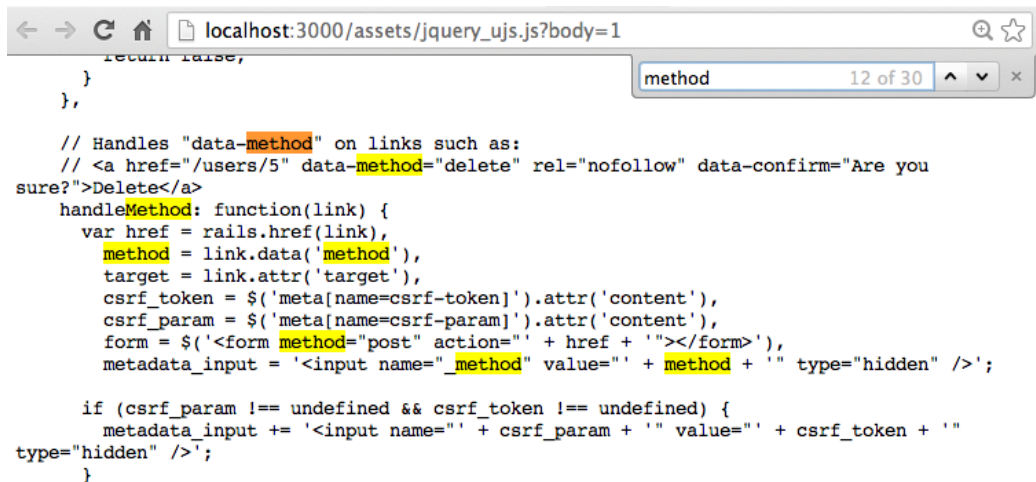


Figure 81: Data Method Delete

```
// Handles "data-method" on links such as:  
// <a href="/users/5" data-method="delete" rel="nofollow" data-confirm="Are yo  
handleMethod: function(link) {  
  var href = rails.href(link),  
      method = link.data('method'),  
      target = link.attr('target'),  
      csrfToken = rails.csrfToken(),  
      csrfParam = rails.csrfParam(),  
      form = $('<form method="post" action="' + href + '"></form>'),  
      metadataInput = '<input name="_method" value="' + method + '" type="hidden
```

You can see handler method that handles 'data-method' on links as shown in the above code snippet.

Step 5

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

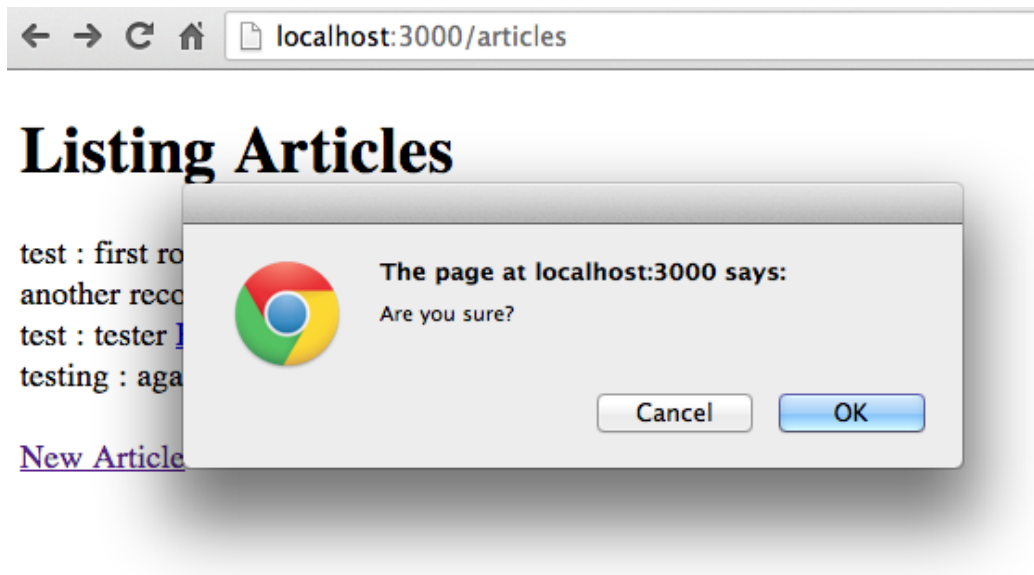


Figure 82: Confirmation Popup

Click 'Cancel' for the confirmation popup window.

Step 6

Define the destroy method in articles controller as follows:

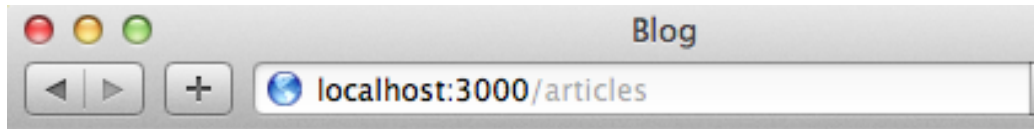
```
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  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 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

record two second row [Edit](#) [Show](#) [Delete](#)
Create new article using a form [Edit](#) [Show](#) [Delete](#)
Create new article using a form second time. [Edit](#) [Show](#) [Delete](#)
[New Article](#)

Figure 83: First Record Deleted

```
Started DELETE "/articles/1" for ::1 at 2016-07-05 20:59:30 -0700
Processing by ArticlesController#destroy as HTML
  Parameters: {"authenticity_token"=>"5xRDtKZR40kc+cGhWielgw", "id"=>"1"}
  Article Load (0.2ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id" = ?
    (0.1ms)  begin transaction
  SQL (0.3ms)  DELETE FROM "articles" WHERE "articles"."id" = ?  [["id", 1]]
    (80.8ms)  commit transaction
Redirected to http://localhost:3000/articles
Completed 302 Found in 95ms (ActiveRecord: 82.2ms)

Started GET "/articles" for ::1 at 2016-07-05 20:59:30 -0700
Processing by ArticlesController#index as HTML
  Rendering articles/index.html.erb within layouts/application
```

```
Article Load (0.2ms)  SELECT "articles".* FROM "articles"  
Rendered articles/index.html.erb within layouts/application (3.1ms)  
Completed 200 OK in 46ms (Views: 43.7ms | ActiveRecord: 0.2ms)
```

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" =>  
<%= javascript_include_tag "application", "data-turbolinks-track" => true %>  
<%= csrf_meta_tags %>  
</head>  
<body>
```

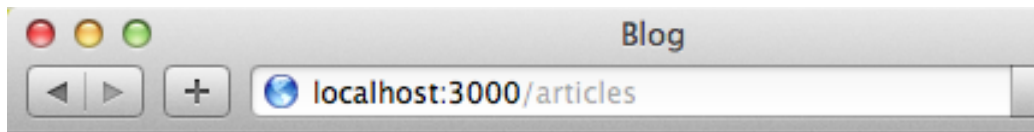
```
<% flash.each do |name, msg| -%>
  <%= content_tag :div, msg, class: name %>
<% end -%>

<%= yield %>

</body>
</html>
```


Step 9

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



Delete success

Listing Articles

Create new article using a form [Edit](#) [Show](#) [Delete](#)

Create new article using a form second time. [Edit](#) [Show](#) [Delete](#)

[New Article](#)

Figure 84: 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’.

```
<body>
    <div class="notice">Delete success</div>
    <h1>Listing Articles</h1>
```

Figure 85: Delete Success Page Source

```
<div class="notice">Delete success</div>
```

You can see the generated html for the notice section.

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 how to eliminate duplication in views.

CHAPTER 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 a new article and edit an 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 86: 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.

CHAPTER 10

Relationships

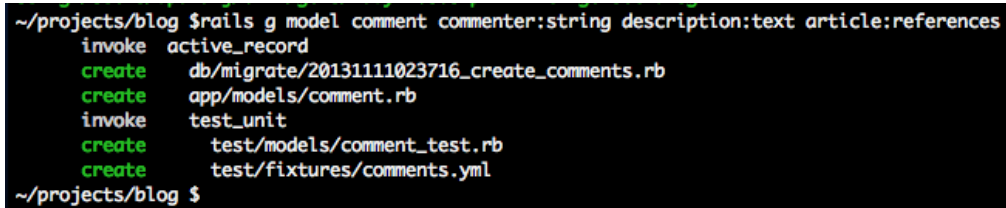
Objective

- 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 87: Generate Comment Model

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

You will see the output:

```
invoke  active_record
create  db/migrate/20160717220101_create_comments.rb
create  app/models/comment.rb
invoke  test_unit
create  test/models/comment_test.rb
create  test/fixtures/comments.yml
```

Step 2

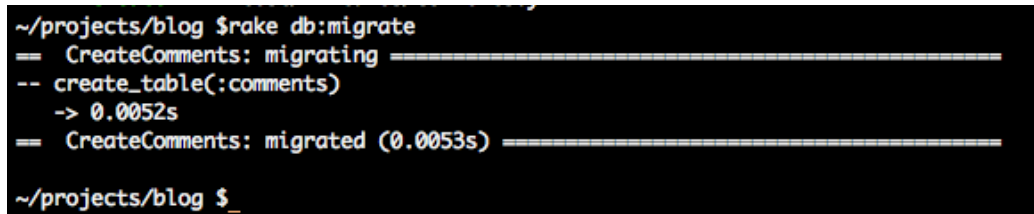
Open the `db/migrate/xyz_create_comments.rb` file in your IDE. You will see the `create_table()` method within `change()` method that takes `comments` symbol as the argument and the description of the columns for the `comments` table.

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 :

```
$ rails db:migrate
```



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

Figure 88: Create Comments Table

```
== 20160717220101 CreateComments: migrating =====
-- create_table(:comments)
   -> 0.0196s
== 20160717220101 CreateComments: migrated (0.0197s) =====
```

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](#)

Step 5

Let's now see the structure of the comments table. In Firefox go to : Tools
→ SQLiteManager

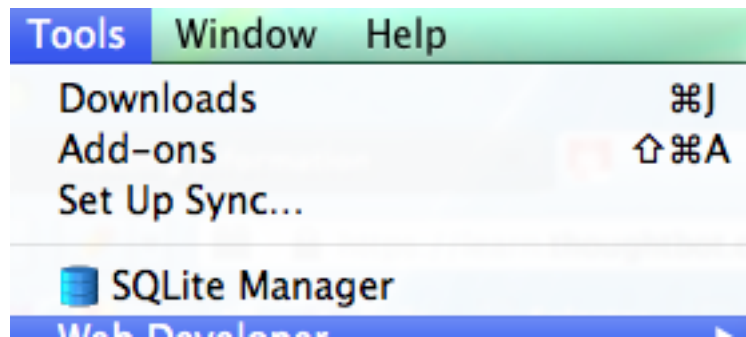


Figure 89: SQLite Manager Firefox Plugin

Step 6

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

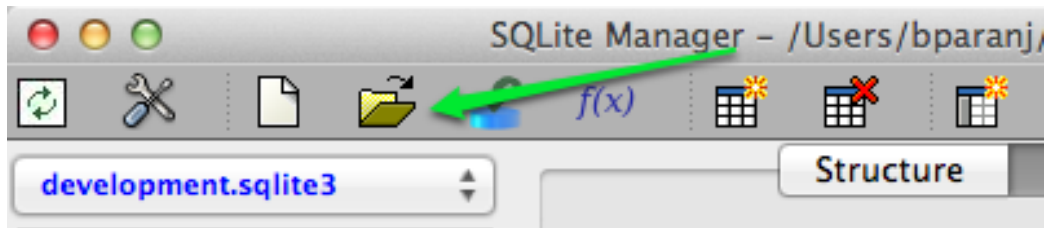


Figure 90: Folder Icon

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

Step 7

Change the file extensions to all files.

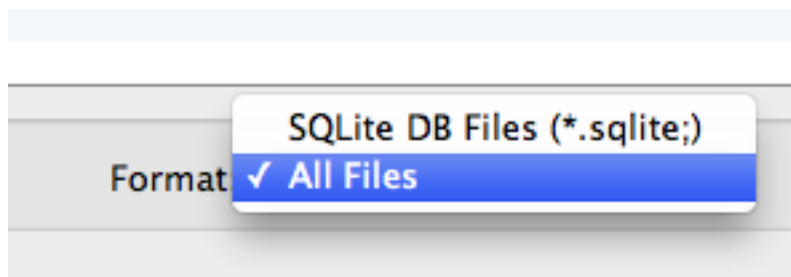


Figure 91: SQLite Manager All Files

Step 8

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

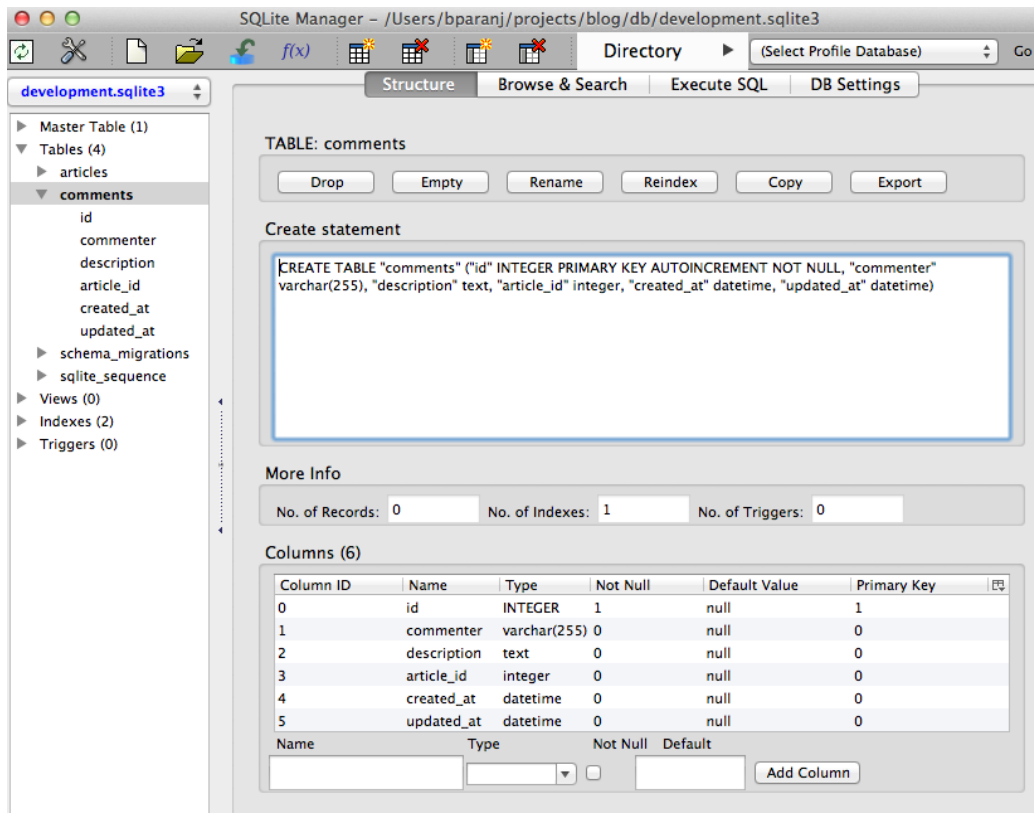


Figure 92: Comments Table Structure

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

You can also use the rails db command to view the tables, schema and work with database from the command line.

```
$ rails db
SQLite version 3.7.7 2011-06-25 16:35:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

To get help:

```
sqlite> .help
```

To list all the databases:

```
sqlite> .databases
seq  name                file
-----
0    main                  /Users/zepho/projects/blog/db/development.sqlite3
```

To list all the tables:

```
sqlite> .tables
ar_internal_metadata  comments
articles              schema_migrations
```

To list the schema for comments table:

```
sqlite> .schema comments
REATE TABLE "comments" ("id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, "commente
CREATE INDEX "index_comments_on_article_id" ON "comments" ("article_id");
```

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.

The `belongs_to` declaration in the model will not create database tables. Since your models are not aware of the database relationships, you need to declare them.

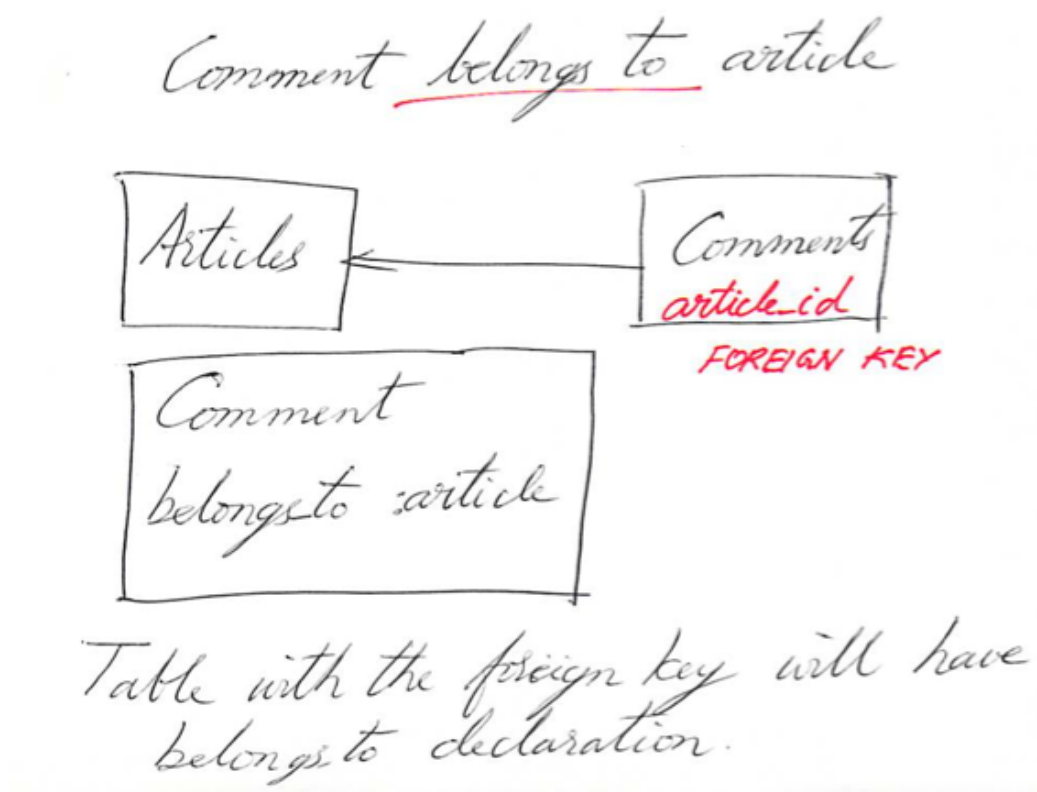


Figure 93: Belongs To Declaration

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.

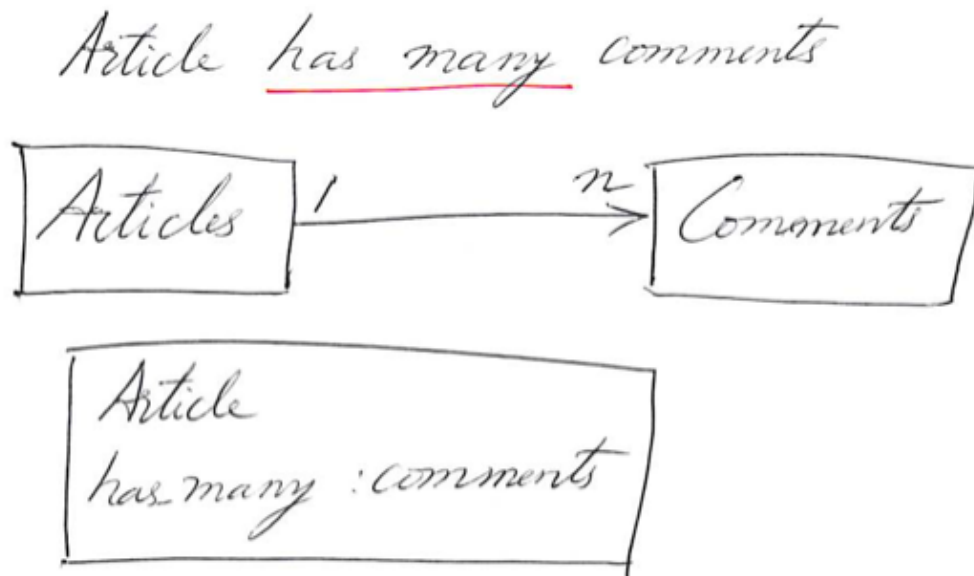


Figure 94: Has Many Declaration

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
```

You will see the output:

```
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
invoke  assets
invoke  coffee
create  app/assets/javascripts/comments.coffee
invoke  scss
create  app/assets/stylesheets/comments.scss
```

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 is a label "Commenter" followed by a single-line text input field. Below that is a label "Description" followed by a larger, multi-line text input field. At the bottom of the form is a button labeled "Create Comment".

Figure 95: 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 96: 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.

```
<form class="new_comment" id="new_comment" action="/articles/5/comments" accept-charset="UTF-8" method="post">
```

Exercise 1

Take a look at the output of **rails routes** command and find out the resource endpoint for the URI pattern `/articles/5/comments` and http method POST combination found in step 15.

Step 16

Run rails routes in the blog directory.

```
zepho-mac-pro:blog5 zepho$ rails routes
      Prefix Verb   URI Pattern                                     Controller#Action
  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
  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
  root GET    /                                              welcome#index
```

Figure 97: Comments Resource Endpoint

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

```
POST    /articles/:article_id/comments(:format)      comments#create
```

Step 17

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

Step 18

Define the create method in `comments_controller.rb` 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 98: Comment Values in Server Log

You can see the comment values in the server log.

```
Started POST "/articles/4/comments" for ::1 at 2016-07-17 15:10:36 -0700
Processing by CommentsController#create as HTML
Parameters: {"authenticity_token"=>"2Gp+UXhpHx", "comment"=>{"commenter"=>"bugs"
No template found for CommentsController#create, rendering head :no_content
Completed 204 No Content in 48ms (ActiveRecord: 0.0ms)
```

Step 20

Copy the entire Parameters hash you see from the server log. Go to Rails console and paste it like this:

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



The screenshot shows a terminal window with three tabs: 'ruby', 'bash', and 'ruby'. The active tab is 'ruby'. The terminal output shows the following commands and results:

```
2.0.0p247 :002 > params = {"utf8"=>"✓", "authenticity_token"=>"6gubQ3YqRqy0RqwhYYyMiy+NEDkNmGbYcj0PXQg8TBg=", "comment"  
=>{"commenter"=>"test", "description"=>"tester"}, "commit"=>"Create Comment", "article_id"=>"5"}  
=> {"utf8"=>"✓", "authenticity_token"=>"6gubQ3YqRqy0RqwhYYyMiy+NEDkNmGbYcj0PXQg8TBg=", "comment"=>{"commenter"=>"test",  
"description"=>"tester"}, "commit"=>"Create Comment", "article_id"=>"5"}  
2.0.0p247 :003 >
```

Figure 99: Parameters for Comment

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



The screenshot shows a terminal window with the following commands and results:

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

Figure 100: Retrieving Comment

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

```
{"commenter"=>"test", "description"=>"tester"}
```

You can extract the `article_id` from the parameters like this:

```
> params['article_id']
```

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 this:

```
@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.

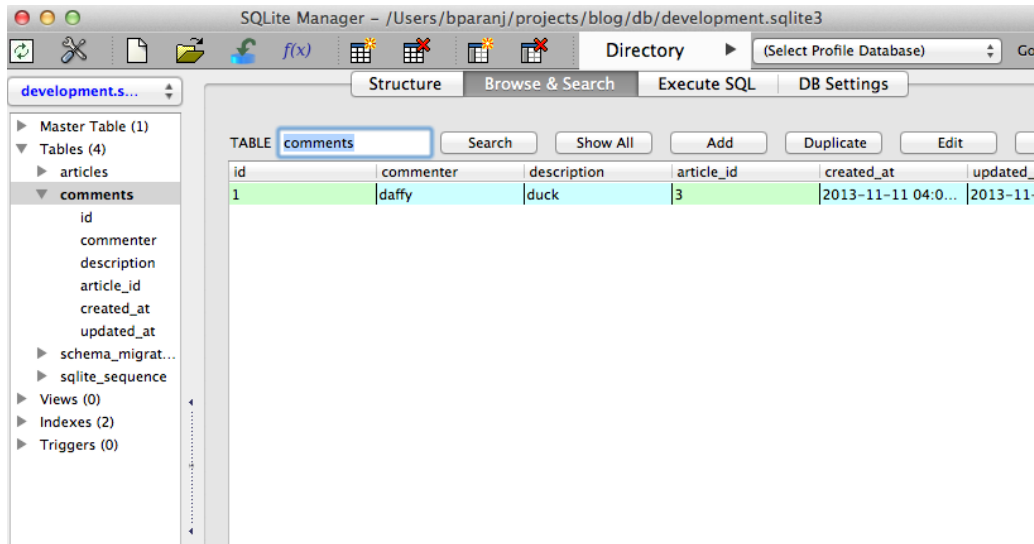


Figure 101: Comment Record in Database

```
Started POST "/articles/4/comments" for ::1 at 2016-07-17 15:13:37 -0700
Processing by CommentsController#create as HTML
  Parameters: {"authenticity_token"=>"oofADG7loSpR07CA==", "comment"=>{"commenter"
Article Load (0.1ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id"
(0.1ms)  begin transaction
SQL (0.4ms)  INSERT INTO "comments" ("commenter", "description", "article_id", "
(54.3ms)  commit transaction
Redirected to http://localhost:3000/articles/4
Completed 302 Found in 61ms (ActiveRecord: 54.9ms)
```

You can now view the record in the MySQLite Manager or Rails db console.
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

Figure 102: Comments For an Article

Summary

We saw how to create parent-child relationship in the database and how to use ActiveRecord declarations in models to handle one to many relationship. 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.

CHAPTER 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:

```
$ rails routes | grep comments
```

Prefix	Verb	URI Pattern	Endpoint
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

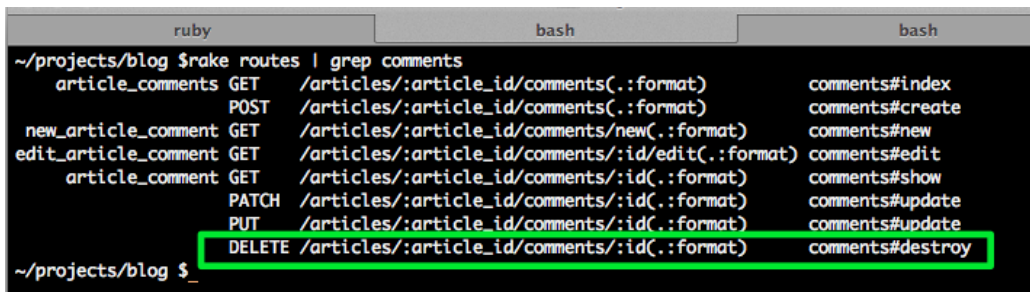
ruby	bash	bash
~/projects/blog \$rake routes grep comments		
article_comments GET	/articles/:article_id/comments(.:format)	comments#index
	POST /articles/:article_id/comments(.:format)	comments#create
new_article_comment GET	/articles/:article_id/comments/new(.:format)	comments#new
edit_article_comment 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 103: Filtered Routes

We are filtering the routes only to the nested routes for comments 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.

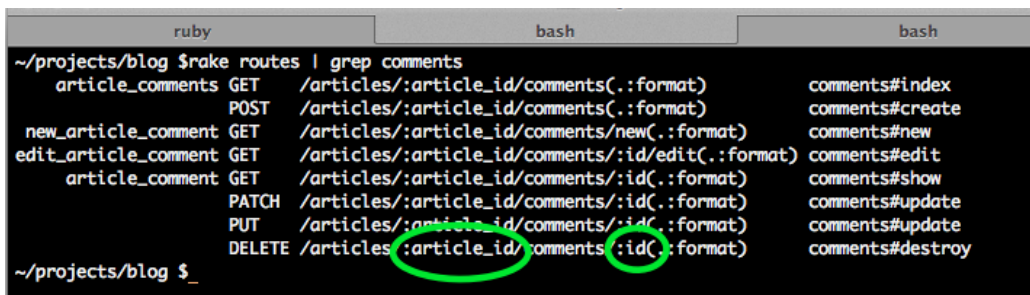


```
~/projects/blog $rake routes | grep comments
  article_comments GET    /articles/:article_id/comments(:format)      comments#index
                  POST   /articles/:article_id/comments(:format)      comments#create
  new_article_comment GET   /articles/:article_id/comments/new(:format)  comments#new
  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
```

Figure 104: Delete URL Helper for Nested Routes

So, we now have:

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



```
~/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
```

Figure 105: 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_path(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_path(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.rb 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. Because we don't want to delete comments that belongs to some other article. 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.

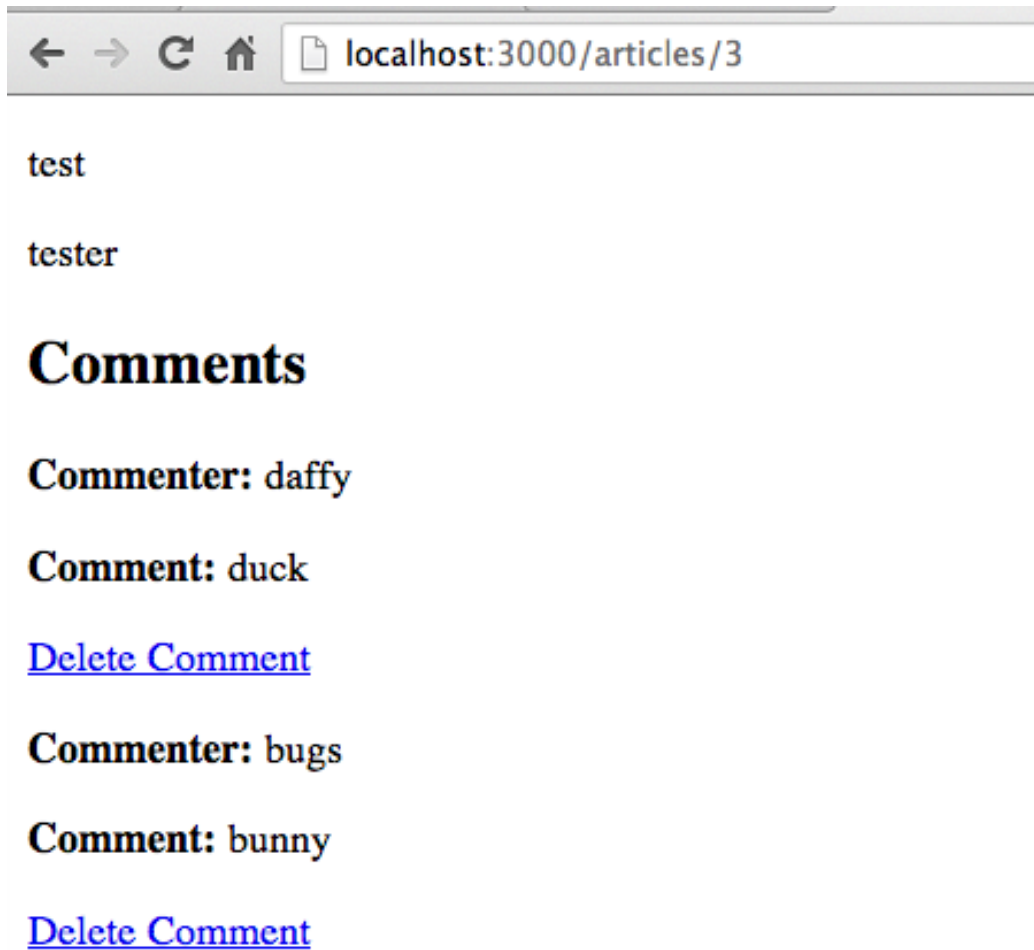


Figure 106: Delete Comment Links

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



Figure 107: URL Error

You will get:

```
undefined method 'article_comment' Did you mean?  article_comment_url
```

if you forget to append the `__path` or `__url` to the `article_comment` Prefix.



Figure 108: Article Instance Variable Error

NameError : undefined local variable or method ‘article’ for #<#<Class:0x007fe>

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

undefined local variable or method ‘article’ for Class:0x007ff
Did you mean? @article

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.

```
Started DELETE "/articles/4/comments/1" for ::1 at 2016-07-17 15:28:28 -0700
Processing by CommentsController#destroy as HTML
Parameters: {"authenticity_token"=>"y4bt1gGyeZK38xZ", "article_id"=>"4", "id"=>}
Article Load (0.1ms)  SELECT  "articles".* FROM "articles" WHERE "articles"."id"=
Comment Load (0.2ms)  SELECT  "comments".* FROM "comments" WHERE "comments"."art
```

(0.1ms) begin transaction
SQL (0.3ms) DELETE FROM "comments" WHERE "comments"."id" = ? [["id", 1]]
(48.3ms) commit transaction
Redirected to <http://localhost:3000/articles/4>
Completed 302 Found in 54ms (ActiveRecord: 49.1ms)

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 109: Refresh Icon

Refresh icon of Firefox Plugin MySQLite Manager.

Exercise 2

Go to articles index page and delete an article that has comments. Now 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 sub-class like this :

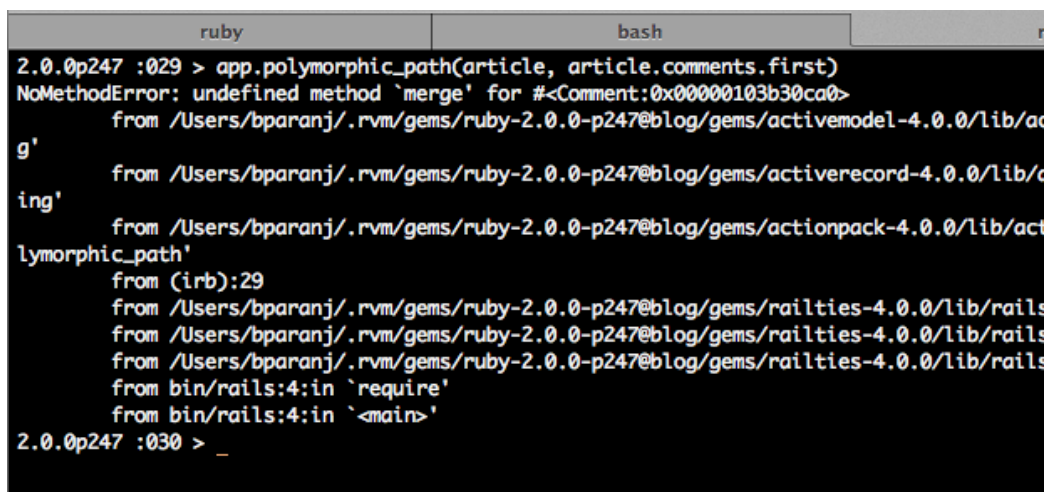
```
class Article < ActiveRecord::Base
  has_many :comments, dependent: :destroy
end
```

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.

Step 10

Let's experiment with `polymorphic_url` method in rails console.

```
a = Loading development environment (Rails 5.0.0)
>> a = Article.first
Article Load (0.2ms) SELECT "articles".* FROM "articles" ORDER BY "articles"."id"
=> #<Article id: 5, title: "Basics of Abstraction", description: "The must know"
>> c = a.comments.first
Comment Load (0.2ms) SELECT "comments".* FROM "comments" WHERE "comments"."article_id" = 5
=> #<Comment id: 5, commenter: "bugs", description: "I agree.", article_id: 5, created_at: "2015-07-27 12:00:00"
>> app.polymorphic_url(a, c)
ArgumentError: wrong number of arguments (given 1, expected 0)
```



```
ruby bash
2.0.0p247 :029 > app.polymorphic_path(article, article.comments.first)
NoMethodError: undefined method `merge' for #<Comment:0x00000103b30ca0>
    from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/activemodel-4.0.0/lib/active_model.rb:100:in `merge'
    from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/activerecord-4.0.0/lib/active_record/connection_adapters/abstract_adapter.rb:100:in `merge'
    from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/actionpack-4.0.0/lib/action_controller/metal/request_forgery_protection.rb:100:in `merge'
    from (irb):29
    from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/railties-4.0.0/lib/rails/console.rb:100:in `merge'
    from /Users/bparanj/.rvm/gems/ruby-2.0.0-p247@blog/gems/railties-4.0.0/lib/rails/console.rb:100:in `merge'
    from bin/rails:4:in `require'
    from bin/rails:4:in `<main>'
2.0.0p247 :030 > _
```

Figure 110: Polymorphic Path Method Error

The `polymorphic_path` method will throw an error when two arguments are passed.

Rails internally uses `polymorphic_path` method with an array containing the parent and child objects to generate the url helper.

```
>> app.polymorphic_url([a, c])
=> "http://www.example.com/articles/5/comments/5"
```

ruby	bash
2.0.0p247 :030 > app.polymorphic_path([article, article.comments.first])	
=> "/articles/5/comments/5"	
2.0.0p247 :031 >	

Figure 111: Polymorphic Path Method

```
>> app.polymorphic_path([a, c])
=> "/articles/5/comments/5"
```

Change the second parameter, url helper in the view to :

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

The link_to will now look like this:

```
<%= link_to 'Delete Comment',
            [@article, comment],
            method: :delete,
            data: { confirm: 'Are you sure?' } %>
```

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

```
a = Article.first
c = a.comments.first
app.polymorphic_url([a, c])
=> "http://www.example.com/articles/4/comments/4"
```

You can learn more about the polymorphic_url by reading the Rails source code. Go to Rails console to find out where it is implemented.

```
ruby > app.respond_to?(:polymorphic_url) => true > app.method(:polymorphic_url).source_location
=> ["/Users/zepho/.rvm/gems/ruby-2.3.1@r5.0/gems/actionpack-5.0.0/lib/action_dispatch.rb", 99]
```

Open the `polymorphic_routes.rb` and look at line 99 in your editor:

```
“ruby module PolymorphicRoutes # Constructs a call to a named RESTful
route for the given record and returns the # resulting URL string. For
example: # # # calls post_url(post) # polymorphic_url(post) # =>
“http://example.com/posts/1” # polymorphic_url([blog, post]) # =>
“http://example.com/blogs/1/posts/1” # polymorphic_url([:admin, blog,
post]) # => “http://example.com/admin/blogs/1/posts/1” # polymor-
phic_url([user, :blog, post]) # => “http://example.com/users/1/blog/posts/1”
# polymorphic_url(Comment) # => “http://example.com/comments” # #
==== Options # # * :action - Specifies the action prefix for the named route:
# :new or :edit. Default is no prefix. # * :routing_type - Allowed values are
:path or :url. # Default is :url. # # Also includes all the options from url_for.
These include such # things as :anchor or :trailing_slash. Example usage #
is given below: # # polymorphic_url([blog, post], anchor: ‘my_anchor’)
# # => “http://example.com/blogs/1/posts/1#my_anchor” # polymor-
phic_url([blog, post], anchor: ‘my_anchor’, script_name: “/my_app”)
# # => “http://example.com/my_app/blogs/1/posts/1#my_anchor”
# # For all of these options, see the documentation for url_for. # #
==== Functionality # # # an Article record # polymorphic_url(record)
# same as article_url(record) # # # a Comment record # polymor-
phic_url(record) # same as comment_url(record) # # # it recognizes
new records and maps to the collection # record = Comment.new #
polymorphic_url(record) # same as comments_url() # # # the class of
a record will also map to the collection # polymorphic_url(Comment) #
same as comments_url() # def polymorphic_url(record_or_hash_or_array,
options = {}) if Hash === record_or_hash_or_array options =
record_or_hash_or_array.merge(options) record = options.delete :id return
polymorphic_url record, options end
```

```
opts    = options.dup
action  = opts.delete :action
type    = opts.delete(:routing_type) || :url
```

```
HelperMethodBuilder.polymorphic_method self,
                                record_or_hash_or_array,
                                action,
                                type,
```

opts

end ““

Summary

In this lesson we learned about nested routes and how to deal with deleting records with children. The current implementation allows anyone to delete records. In the next lesson we will restrict the delete functionality to blog owner.

CHAPTER 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.

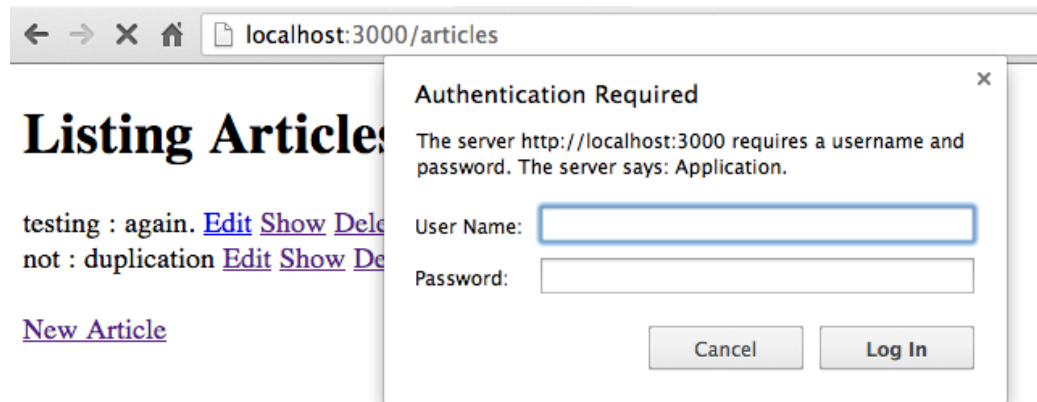


Figure 112: URL Error

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.

Exercise 2

In the rails console do this:

```
> ApplicationController.respond_to?(:http_basic_authenticate_with)
=> true
```

```
> ApplicationController.method(:http_basic_authenticate_with).source_location  
=> ["/Users/zepho/.rvm/gems/ruby-2.3.1@r5.0/gems/actionpack-5.0.0/lib/action_controllers
```

Go to the file shown in the output and line 68. Read the code.

Summary

This completes our quick tour of Rails 5. 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.

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.

B. Troubleshooting

1. Use rails console to experiment.
2. To inspect a variables in views you can use `debug`, `to_yaml` and `inspect`.

```
<%= debug(@article) %>
```

will display the `@article` object in YAML format.

The `to_yaml` can be used anywhere (not just views). You can do a query in Rails console and call `to_yaml` on an article object.

```
article = Article.first
article.to_yaml
```

The `inspect` method is handy to display values in arrays and hashes.

```
a = [1,2,3,4]

p a.inspect
```

If you customize the `to_s` method in your classes then the `inspect` method will use your `to_s` method to create a human friendly representation of the object.

```
class Car

  def to_s
    "I am a car"
  end
end

c = Car.new

print c
```

3. You can use `logger.info` in the controller to log messages to the log file. In development log messages will go to `development.log` in log directory.

```
logger.info "You can log anything here #{@article.inspect}"
```

To use the logger in model, you have to do the following:

```
Rails.logger.info "My logging goes here"
```

4. Using `tail` to view development log file.

Open a new tab in the terminal (On Mac `Command+T` opens a new tab on an existing open terminal), go the rails project blog directory and type the following command:

```
$ tail -f log/development.log
```

5. View source in the browser. For example: Checking if path to images are correct.
6. Use rails `dbconsole`
7. Firebug Firefox plugin, Chrome Dev Tools or something equivalent
8. Debugger in Rubymine is simply the best debugger. JetBrains updates fixes any issues with Ruby debugging gems and provides a well integrated IDE for serious development.
9. Useful plugins:
 - [Rails Footnotes](#)
 - [Rails Panel - Chrome Extension for Rails Development](#)
10. Spring can cause headaches. If the changes you make is not getting picked up, exit all Rails console sessions and restart the server. This seems to fix problems.

C. FAQ

1. Adding a new source to gem.

```
$ gem sources -a http://gems.github.com
```

2. Suppress installing rdoc for gems. For example to install passenger gem without any rdoc or ri type:

```
$ gem install passenger -d --no-rdoc --no-ri
```

3. How to upgrade gems on my system?

```
$ gem update -system
```