**CREATE A NEW RAILS PROJECT**

3.1 Installing rails

1. Verify you have a current version of Ruby installed:

$ ruby –v

1. Verify that SQLite3 is correctly installed and in your PATH:

$ sqlite3 --version

The program should report its version.

1. To install Rails, use gem install, the command provided by RubyGems:

$ gem install rails

1. To verify you have everything installed correctly, you should be able to run the following:

$ rails --version

3.2 Creating the Blog Application

1. Use the new application generator to create everything necessary to start working on a new task:

$ rails new blog

🡪 This will create a Rails application called Blog in a blog directory, and install the gem dependencies that are already mentioned in Gemfile using bundle install.

To see all command line options that the Rails application builder accepts, run: rails new –h

1. Switch to the blog application folder:

$ cd blog

**4) Hello, Rails!**

* 1. Starting up the Web Server

1. To see the Rails app, start a web server on your dev machine. Run the following in the blog directory:

$ bin/rails server

🡪 This will fire up Puma, a web server distributed with Rails by default. To see the app in action, go to <http://localhost:3000> in the browser

4.2 Get Rails to say “Hello”

1. To do this, you need to create a *controller* and a *view*
   1. The *controller* receives specific requests for the app – this is where information is collected (then, is displayed by the view).
   2. *Routing* decides which controller receives which requests
   3. There is often more than one route to each controller, and different routes can be served by different *actions*.
   4. Each action’s purpose is to collect info and provide it to a view, which displays this info in human readable format.
      1. By default, view templates are written in a language called eRuby (embedded Ruby) which is processed by the request cycle in Rails before being sent to the user.
2. To create a new controller, run the “controller” generator and tell it you want a controller called “Welcome” with an action called “index”:

$ bin/rails generate controller Welcome index

🡪 Rails will create several files and a route for you.

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 scss

create app/assets/stylesheets/welcome.scss

1. Open the app/views/welcome/index.html.erb file in your text editor. Delete all of the existing code in the file, and replace it with:

<h1>Hello, Rails!</h1>

4.3 Setting the App Home Page

Now that we’ve made the controller and view, we need to tell Rails when we want “Hello, Rails!” to show up; in our case, we want it to show up when we navigate to the root URL of the site, <http://localhost:3000> - right now, “Welcome Aboard” is occupying that spot (I think this is ‘Yay! You’re on Rails!’ 🡪 is that right?)

1. To tell Rails where the home page is located:

* Open the file config/routes.rb in the text editor (Sublime, in my case) – the config sub-folder is a direct sub-folder of ‘blog’
  + This is the application’s routing file, which holds entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions.
* Edit the file by adding the line of code root ‘welcome#index’ after get ‘wecome/index’ so that it looks like:

Rails.application.routes.draw do

get 'welcome/index'

root ‘welcome#index’

# For details on the DSL available within this file, see http://guides.rubyonrails.org/routing.html

end

* root ‘welcome#index’ tells Rails to map requests to the root of the application, to the welcome controller’s index action
* get ‘welcome/index’ tells Rails to map requests to <http://localhost:3000/welcome/index> to the welcome controller’s index action (this was created when I ran the controller action earlier (bin/rails generate controller Welcome index)
* NOW, the web server should be running the “Hello, Rails!” message I put in to app/views/welcome/index.html.erb, indicating that this new route is going to WelcomeController’s index action and is rendering the view correctly.
* **Q: WHAT WAS THE ROOT PREVIOUSLY? WHERE WAS THE ORIGINAL ‘YAY RAILS ETC.’ MESSAGE COMING FROM?**

**5) Getting Up and Running**

Now that the controller, an action and a view have been created, I’m going to create a new **resource** in the Blog application.

* A **resource** is a term used for a collection of similar objects, such as articles, people or animals.
  + You can create, read, update and destroy items for a resource 🡪 these operations are referred to as *CRUD* operations
* Rails provides a *resources* method which can be used to declare a standard *REST resource*. You need to add the *article resource* to the config/routes.rb so the file will look as follows:

Rails.application.routes.draw do

get 'welcome/index'

resources :articles

root 'welcome#index'

# For details on the DSL available within this file, see http://guides.rubyonrails.org/routing.html

end

* When we run bin/rails routes, we’ll see that it has defined routes for all the standard RESTful actions. 🡪 Rails can infer the singular form article and make meaningful use of the distinction.

Jessicas-MacBook-Pro:blog jessicafisher$ bin/rails routes

       Prefix Verb   URI Pattern                  Controller#Action

welcome\_index GET    /welcome/index(.:format)     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

         root GET    /                            welcome#index

**5.1 Laying down the groundwork** – adding the ability to create new articles in your app and be able to view them – the “C” and “R” from CRUD: create and read.

1. Pick a place within the app to create a new article: /articles/new would be great; with the route already defined, requests can now be made to /articles/new in the app.
2. If we navigate to <http://localhost:3000/articles/new> we will see a routing error.
   1. Error occurs because the route needs to have a controller defined to serve the request. The error reads:  **uninitialized constant ArticlesController**
   2. So we need to a create a controller called ArticlesController. Do this by running the command:

$ bin/rails generate controller Articles

Now, within a new controller, which can be found in app/controllers/articles\_controller.rb, will be generated – it’s fairly empty.

class ArticlesController < ApplicationController

end

🡪 A controller is just a class that’s defined to inherit from ApplicationController.

🡪 Inside this class we’ll define methods that will become the actions for this controller; these actions will perform CRUD operations on the articles within our system.

There are public, private and protected methods in Ruby, but only public methods can be actions for controllers.

1. Now, at localhost:3000/articles/new, we’ll get a new error:

Unknown action

The action 'new' could not be found for ArticlesController

🡪 indicates that Rails cannot find the new action inside the ArticlesController that was generated 🡪 since when controllers are generated in Rails they are empty by default, unless we tell them the desired actions during the generation process

1. To manually define an action inside a controller, you need to define a new method inside the controller.

🡪 Open app/controllers/articles\_controller.rb and inside the ArticlesController class, define the new method so that the controller now looks like:

class ArticlesController < ApplicationController

def new

end

end

a. A new error will pop up:

**ActionController::UnknownFormat in ArticlesController#new**

**ArticlesController#new is missing a template for this request format and variant. request.formats: ["text/html"] request.variant: [] NOTE! For XHR/Ajax or API requests, this action would normally respond with 204 No Content: an empty white screen. Since you're loading it in a web browser, we assume that you expected to actually render a template, not… nothing, so we're showing an error to be extra-clear. If you expect 204 No Content, carry on. That's what you'll get from an XHR or API request. Give it a shot.**

This error message pops up because Rails expects plain actions like this one to have views associated with them to display their information. With no view available, Rails will raise an exception.

Let’s decode the error message.

🡪 Part 1: indicates which template is missing: it’s the articles/new template.

* + Rails will first look for this template
  + If it’s not found, it will attempt to load a template called application/new
  + It looks for a template because ArticlesController inherits from ApplicationController

🡪 Part 2: contains request.formats - which specifies the format of template to be served in response.

* + It’s set to text/html as we requested this page via browser, so Rails is looking for an HTML template
  + Request.variant specifies what kind of physical devices would be served by the response and helps Rails determine which template to use in response; it’s empty here because no info has been provided

🡪 SO. The simplest template that would work in this case would be one located at app/views/articles/new.html.erb

* + The extension of the file name: the first extension is the *format* of the template, and the second extension is the *handler* that will be used to render the template
  + Rails is attempting to find a template called articles/new within app/views for the app. The format for this template can only be html and the default handler for HTML is erb.
    - \*\* Rails uses other handlers for other formats.
      * Builder handler is used to build XML templates
      * Coffee handler uses CoffeeScript to build JavaScript templates
    - Since I want to create a new HTML form, I’m going to use the ERB language which is designed to embed Ruby in HTML.
      * The file should therefore be called articles/new.html.erb and needs to be located inside the app/views directory of the app

1. Now, I’m creating a new file at app/views/articles/new.html.erb and writing the following inside:

<h1>New Article</h1>

🡪 Now, when we refresh the page at <http://localhost:3000/articles/new> it has a title, New Article.

**5.2 The first form**

We use a *form builder* to create a form within this template. The primary form builder for Rails is provided by a helper metho called form\_for. To use this method, add the following code into app/views/articles/new.html.erb:

<%= form\_for :article do |f| %>

<p>

<%= f.label :title %><br>

<%= f.text\_field :title %>

</p>

<p>

<%= f.label :text %><br>

<%= f.text\_area :text %>

</p>

<p>

<%= f.submit %>

</p>

<% end %>

* WHEN CALLING form\_for, I'M PASSING IT AN IDENTIFYING OBJECT FOR THIS FORM. IN THIS CASE, IT'S THE SYMBOL :article. THIS TELLS THE form\_for HELPER WHAT THE FORM IS FOR.
* Inside the block for this method, the FormBuilder object - represented by f - is used to build two labels and two text fields, one each for the title and text of an article.
* A call to submit on the f object will create a submit button for the form.
* Issue: when you inspect the HTML that’s generated, by viewing the source of the page, we see that the action attribute for the form points at /articles/new. 🡪 Q: how do I do this??
  + Causes a problem bc this route goes to the page that you’re on at the moment, and that route should only be used to display the form for a new article
    - So the form needs to use a different URL in order to go somewhere else. Can do this with the :url option of form\_for
    - Typically in Rails, the action that’s used for new form submissions like this is called “create” and so the form should be pointed to that action.
* So, we edit the form\_for line within app/views/articles/new.html.erb so that it looks like:

<%= form\_for :article, url: articles\_path do |f| %>

* Now, the articles\_path helper is passed to the :url option. To see what Rails will do with this, look back at the output of bin/rails routes:

Jessicas-MacBook-Pro:blog jessicafisher$ bin/rails routes

       Prefix Verb   URI Pattern                  Controller#Action

welcome\_index GET    /welcome/index(.:format)     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

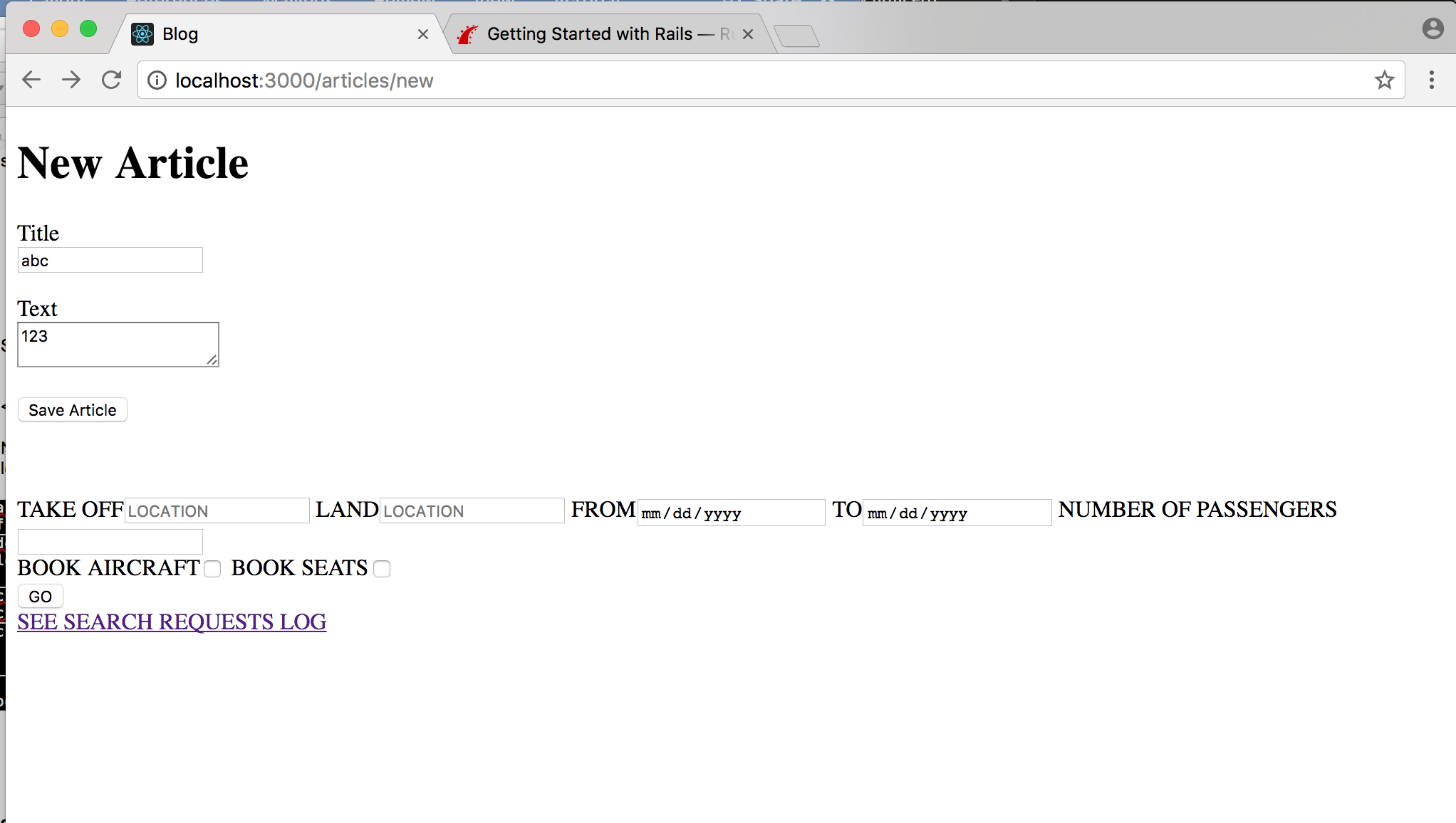
      article GET    /articles/:id(.:format)      articles#show

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

              PUT    /articles/:id(.:format)      articles#update

              DELETE /articles/:id(.:format)      articles#destroy

         root GET    /                            welcome#index



**5.3 Creating articles**

The articles\_path helper tells Rails to point the form to the URI pattern associated with the articles prefix; the form will (by default) send a POST request to that route. 🡪 This is associated with the create action of the current controller, the ArticlesController.

With the form and the associated route defined, I’ll be able to fill in the form and then click the submit button to begin the process of creating a new article. When you submit the form, the following error pops up:

Unknown action

The action 'create' could not be found for ArticlesController

Now, I need to create the create action within the ArticlesController for this to work.

**5.3 Creating Articles**

To make the “Unknown action” go away, define a create action within the ArticlesController class in app/controllers/articles\_controller.rb, underneath the new action:

class ArticlesController < ApplicationController

def new

end

# Adding the following to define a create action within the ArticlesController class:

def create

end

end

* If you re-submit the page now, there’s no change to the page. This is bc Rails by default returns 204 No Content response for an action if we don’t specify what the response will be. We just added the create action but didn’t specify anything about how the response should be; the create action should save our new article to the database.
* When the form is submitted, the fields of the form are sent to Rails as parameters. These parameters can then be referenced inside the controller actions, typically to perform a particular task. To see what these parameters look like, change the create action to:

def create

render plain: params[**:article**].inspect

end

🡪 here, the render method takes a simple hash with a key of :plain and value of params[:article].inspect.

* + The params method is the object which represents the parameters (or fields) coming in from the form.
  + The params method returns an ActionController::Parameters object, which allows you to access the keys of the hash using strings or symbols.

**Ruby - Hashes**. A Hash is a collection of key-value pairs like this: "employee" = > "salary". It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index. (Reference URL: <https://www.tutorialspoint.com/ruby/ruby_hashes.htm)>

Params method:

Ex) URL: <http://www.example.com/?username=dhh&email=dhh@email.com>

* + - params[:username] is “dhh”
    - params[:email] is [dhh@email.com](mailto:dhh@email.com)

If you re-submit the form again, you’ll see the following:

<ActionController::Parameters {"title"=>"", "text"=>""} permitted: false>

The action now displays the parameters for the article that are coming in from the form; however, while you can see the parameters, you can’t see what’s being done with them.

**5.4 Creating the Article model**

* Models in Rails use a singular name, and their corresponding database tables use a plural name.
* Rails provides a generator for creating models. To create the new model, run this command in terminal:

$ bin/rails generate model Article title:string text:text

🡪 This command tells Rails we want an Article model, together with a *title* attribute of type string, and a *text* attribute of type text.

* + Those attributes are automatically added to the *articles* table in the database and mapped to the *Article* model

🡪 Rails responded by creating a bunch of files. For now, we are interested in app/models/article.rb and db/migrate/ 20170626124808\_create\_articles.rb. The latter is responsible for creating the database structure

* + Active Record automatically maps column names to model attributes, so I don’t have to declare attributes inside Rails models (since that’s automatically done by Active Record)

**5.5 Running a Migration**

bin/rails generate model created a database migration file inside the db/migrate directory

* Migrations are Ruby classes designed to make it siple to create and modify database tables
* Rals uses rake commands to run migrations, and it’s possible to undo a migration after its been applied to your database
* Migration filenames include a timestamp to ensure they’re processed in the order they were created

Within db/migrate/ 20170626124808\_create\_articles.rb, we’ll find the following:

class CreateArticles < ActiveRecord::Migration[5.1]

def change

create\_table :articles do |t|

t.string :title

t.text :text

t.timestamps

end

end

end

* Here, the class CreateArticles is inheriting from Migration, a sub-class of the ActiveRecord class.
* Def change refers to us changing the structure of the database
* …this migration creates a method named change which will be called when I run this migration.
  + The action defined in this method is also reversible, so Rails knows how to reverse the change made by this migration, in case I want to reverse it later.
  + Running this migration will create an articles table with one string column and a text column
    - It also creates two timestamp fields to allow Rails to track article creation and update times.
* Use a bin/rails command to run the migration:

$ bin/rails db:migrate

🡪 db: migrate is like running a ‘seed file’ – it runs a migration file

* + Rails will execute this migration command and tell you it created the Articles table.
  + Because I’m working in the development environment by default, the command will apply to the database defined in the development section of my config/database.yml file.
    - \*\* TO EXECUTE MIGRATIONS IN ANOTHER ENVIRONMENT, FOR INSTANCE IN PRODUCTION, I MUST PASS IT WHEN INVOKING THE COMMAND: bin/rails db:migrate RAILS\_ENV=production

**5.6 Saving data in the controller**

* Back in ArticlesController, we need to change the create action to use the new Article model to save the data in the database.
  + Open app/controllers/articles\_controller.rb and change the create action to the following:

def create

@article = Article.new(params[:article])

@article.save

redirect\_to @article

end

and delete the former content:

# Adding the following to define a create action within the ArticlesController class:

def create

# when the form is submitted, the form fields are sent to Rails as parameters. These parameters can be referenced inside the controller actions, typically to perform a particular task. To see what they look like, change the create action to:

render plain: params[:article].inspect

end

🡪 What’s going on here?

* + Every Rails model can be initialized with its respective attributes, which are automatically mapped to the respective database columns.
    - In the first line we do just that – params[:article] contains the attributes we’re interested in.
    - Then, @article.save is responsible for saving the model in the database
    - Last, we redirect the user to the show action, which we’ll define later
    - The A in Article.new is capitalized because in this context, we are referring to the class named Article defined in app/models/article.rb.
      * Class names in Ruby must begin with a capital letter
    - @article.save returns a Boolean indicating whether the article was saved or not
* Now, after adding the new create text into articles\_controller.rb, I’m seeing an error msg:

**ActiveModel::ForbiddenAttributesError in ArticlesController#create**

**ActiveModel::ForbiddenAttributesError**

Extracted source (around line **#8**):

|  |  |
| --- | --- |
| 6  7  8  9  10  11 | ## to save data in the database  def create  @article = Article.new(params[:article])  @article.save  redirect\_to @article |

🡪 that’s because of the Rails security measure strong parameters, which requires us to tell Rails exactly which parameters are allowed into our controller actions.

* The ability to grab and automatically assign all controller parameters to the model in one shot allows malicious use – if a request to the server was crafted to look like a new article form submit but also included extra fields with values that violated the app’s integrity, they would be ‘mass assigned’ into the model and then into the database along with the good stuff, and could potentially break the app, etc.
* We need to whitelist our controller parameters to prevent wrongful mass assignment 🡪 so we want to both allow and require the *title* and *text* parameters for valid use of create. The syntax for this introduces *require* and *permit*.
  + The change will require one line in the *create* action:

@article = Article.new(params.require(:article).permit(:title, :text))

🡪 And this is often factored out into its own method so it can be reused by multiple actions in the same controller; for ex, *create* and *update*.

🡪 Above and beyond mass assignment issues, the method is often made *private* to make sure it can’t be called outside its intended context.

The result looks like this:

def create

@article = Article.new(article\_params)

@article.save

redirect\_to @article

end

private

def article\_params

params.require(:article).permit(:title, :text)

end

end

🡪 So, we replace the following (outdated) text:

def create

@article = Article.new(params[:article])

@article.save

redirect\_to @article

end

end

**5.7 Showing Articles**

* Now, when we submit an article, we get this error message:

**Unknown action**

**The action 'show' could not be found for ArticlesController.**

* + This is Rails complaining about not finding the show action; we need to add the show action before proceeding.
  + As seen in the output of bin/rails routes, the route for *show* action is as follows:

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

🡪 The special syntax :id tells rails that this route expects an :id parameter, which in our case will be the id of the article

* As we did before, we need to add the **show** action in app/controllers/articles\_controller.rb and its respective view.
  + Frequent practice is to place the standard CRUD actions in each controller in the following order: index, show, new, edit, create, update and destroy.
    - Can use any method you want, but these are public methods; as mentioned earlier, they must be placed before declaring *private* visibility in the controller.
* Given that, let’s add the *show* action, as follows:

class ArticlesController < ApplicationController

def show

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

end

def new

end

etc.

* + Note: we use Article.find to find the article we’re interested in, passing params[:id] to get the :id parameter from the request.
  + We also use an instance variable (prefixed with @) to hold a reference to the article object.
    - Rails will pass all instance variables to the view.
* Now, create a new file app/views/articles/show.html.erb with the following content:

<p>

<strong>Title:</strong>

<%= @article.title %>

</p>

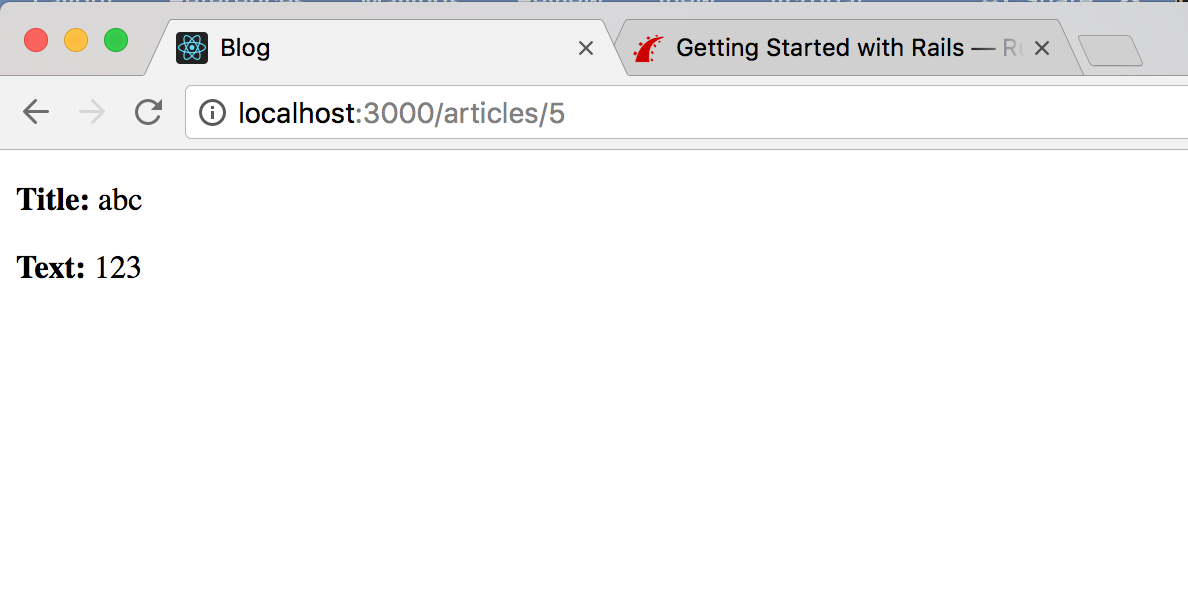
<p>

<strong>Text:</strong>

<%= @article.text %>

</p>

🡪 This change should allow for the creation of new articles. – see <http://localhost:3000/articles/new>



**5.8 Listing all articles**

* We still need a way to list all our articles: the route for this as per output of *bin/rails routes* is:

articles GET /articles(.:format) articles#index

* Add the corresponding *index* action for that route inside the ArticlesController in the app/controllers/articles\_controller.rb file. When we write an *index* action, the usual practice is to place it as the first method in the controller.

class ArticlesController < ApplicationController

def index

@articles = Article.all

end

def show

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

end

* Then add the view for this action, located at app/views/articles/index.html.erb:

<h1>Listing articles</h1>

<table>

<tr>

<th>Title</th>

<th>Text</th>

</tr>

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

<tr>

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

<td><%= article.text %></td>

<td><%= link\_to 'Show', article\_path(article) %></td>

</tr>

<% end %>

</table>

🡪 Now, at <http://localhost:3000/articles> there’s a list of articles that have been created.

**5.9 Adding links**

At this point, I can create, show and list articles. Next, add links to navigate through pages.

**Q: How to make these links work in new tabs, rather than in the same tab within the browser??**

* Open app/views/welcome/index.html.erb and modify it as follows:

<h1>Hello, Rails!</h1>

<%= link\_to 'My Blog', controller: 'articles' %>

* + The link\_to method is one of Rails’ built-in view helpers. It creates a hyperlink based on text to display and where to go – in this case, to the path for articles.
* Next, we’ll add links to other views, starting with adding this “New Article” link to app/views/articles/index.html.erb, placing it above the <table> tag:

<%= link\_to ‘New article’, new\_article\_path %>

**🡪** This link will allow you to bring up the form that lets you create a new article.

* To go back to the *index* action, add another link in app/views/articles/new.html.erb:

<%= link\_to 'Back', articles\_path %>

* Last, add a link to the app/views/articles/show.html.erb template to go back to the index action as well, so that people who are viewing a single article can go back and view the whole list again (adding just the last line included here):

<p>

<strong>Title:</strong>

<%= @article.title %>

</p>

<p>

<strong>Text:</strong>

<%= @article.text %>

</p>

<%= link\_to 'Back', articles\_path %>

* + If you want to link to an action in the same controller, you don’t need to specify the :controller option; Rails will use the current controller by default.
  + In development mode (what I’m working in now), Rails reloads the application with every browser request, so there’s no need to stop and restart the web server when a change is made.

**5.10 Adding Some Validation**

* The model file, app/models/article.rb is as simple as it can get:

class Article < ApplicationRecord

end

* + The Article class inherits from ApplicationRecord. ApplicationRecord inherits from ActiveRecord::Base which supplies a great deal of functionality to the Rails model for free, including basic database (CRUD) operations, data validation, and sophisticated search support and the ability to relate multiple models to one another.
* Rails includes methods to help you validate the data that you send to models. Open the app/models/article.rb file and edit it:

class Article < ApplicationRecord

end

..becomes the following post-edit:

class Article < ApplicationRecord

validates :title, presence: true,

length: { minimum: 5 }

end

🡪 These changes ensure that all articles have a title that’s at least five characters long. Rails can validate a variety of conditions in a model, including the presence or uniqueness of columns, their format, and the existence of associated objects.

* With the validation now in place, when you call @article.save on an invalid article, it will return false. If you open app/controllers/articles\_controller.rb again, you’ll notice we don’t check the result of calling @article.save inside the create action.
* If @article.save fails in this situation, we need to show the form back to the user. To do this, change the new and create actions inside app/controllers/articles\_controller.rb to the following:

def new

@article = Article.new

end

# Now we are changing the create action to use the new Article model

## to save data in the database

def create

@article = Article.new(article\_params)

if @article.save

redirect\_to @article

else

render 'new'

end

end

private

def article\_params

params.require(:article).permit(:title, :text)

end

end

----------

From, previously:

def new

end

# Now we are changing the create action to use the new Article model

## to save data in the database

def create

@article = Article.new(article\_params)

@article.save

redirect\_to @article

end

private

def article\_params

params.require(:article).permit(:title, :text)

end

* The new action is now creating a new instance variable called @article – we’ll see why in a few moments.

Inside the **create** action we use **render** instead of **redirect\_to** when **save** returns **false**. The **render** method is used so that the @article object is passed back to the **new** template when it is rendered.

* + This rendering is done within the same request as the form submission, whereas the **redirect\_to** will tell the browser to issue another request.
  + If you reload <http://localhost:3000/articles/new> and try to save an article without a title, Rails will send you back to the form
    - That’s not that useful without telling the user that something went wrong – to do that, modify the app/views/articles/new.html.erb to check for error messages:

…add after <%= form\_for :article, url: articles\_path do |f| %> :

<% if @article.errors.any? %>

<div id="error\_explanation">

<h2>

<%= pluralize(@article.errors.count, "error") %> prohibited

this article from being saved:

</h2>

<ul>

<% @article.errors.full\_messages.each do |msg| %>

<li><% = msg %></li>

<% end %>

</ul>

</div>

<% end %>

* + Here, we are checking if there are any errors with @article.errors.any?, and in that case we show a list of all errors with @article.errors.full\_messages.
  + Pluralize is a rails helper that takes a number and a string as its arguments. If the number is greater than one, the string will automatically be pluralized.
  + We added @article = Article.new in the ArticlesController because otherwise @article would be nil in our view, and calling @articles.errors.any? would throw an error.
* Rails automatically wraps fields that contain an error with a div with class field\_with\_errors. You can define a css rule to make them standout.
* Now, I’ll see an error message when saving an article without a title when attempting to do just that on the new article form <http://localhost:3000/articles/new>:

**New Article**

Top of Form

**2 errors prohibited this article from being saved:**

* Title can't be blank
* Title is too short (minimum is 5 characters)

**5.11 Updating Articles**

We’ve covered the “CR” part of CRUD – now I’m focusing on the “U” part, updating articles.

Bottom of Form