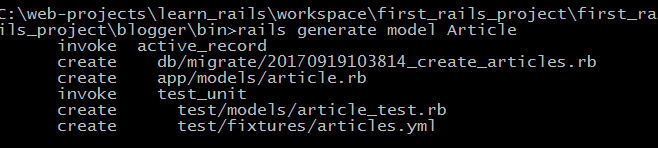
Making a Blog with rails

Saved in C:\web-projects\learn\_rails\workspace\first\_rails\_project\first\_rails\_project\blogger

1. I initiated a repo on github and wrote an initial read.me file
2. Ran rails new blogger- this command creates a basic rails app which I can personalise
3. All commands are now run from the bin directory
4. Rails server- this command (in a new terminal) will run a server from my machine allowing me to open my rails app with my browser to see the welcome to rails page at local host 3000.

**Setting up the Article Model**

1. The blog will contain articles so I need to create a model which will be able to interact with the articles in the database. To do this I use rails generate model Article. This command will create all the files I need to interact with the articles in the database.

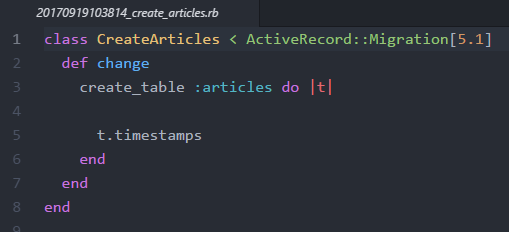


The db migrate (first line) is a database migration. It has a time stamp. A migration is able to alter our database for us. Each migration creates a new version of the database, altering tables, columns or entries. This migration will be able to create the articles table in the database.

app/models/article.rb- this ruby file will contain the code for the model

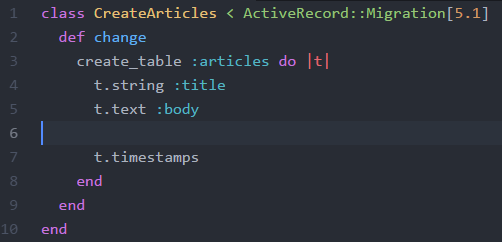
The last two files are associated with testing

**The Article Migration**



This is the current file for the migration. It is named with a timestamp which allows the migrations to be kept in chronological order. It has only one method-change. Within the change method create\_table has been called automatically. This is a method which will create a table! The generator has passed the create\_table method the symbol :articles as a parameter. This will create a table called Articles. There is then a block which assigns the variable t to the table which is created.

,t is important because we call methods on t to create the columns in the articles table. We will need a column called title and a column named body to hold the data from our articles. The title will be defined as a string and the body is defined as text. “Text” is data type which the rails adapters can change to whatever type our database will understand without us having to worry about it. This could be varchar or text depending on the database used(postgres/sqlite).



We edit the change method to include t.string :title and t.text :body. This will create our columns with the correct titles and data types. Rails also automatically makes two columns using t.timestamps. These columns are titled “created\_at“ and “updated\_at”. These record the times that articles are uploaded and when they are subsequently updated. Rails will do the recording for us.

We are now ready to run the migration (to create our table in the database). We do this with the command, rake rb:migrate. Rake is used to carry out maintainance functions and testing. The command says, “look in your functions for databases and run the migrate function”. The migrate action will then find all the migrations in db/migrations folder. It then looks at a table in the database which says which migrations haven’t been run. It will then run the migrations which haven’t been run yet.

In our case CreateArticles was run and made a table named articles.

**WORKING WITH A MODEL IN THE CONSOLE-Side step 😉**

If we want to we can access all parts of our app with the command line. If we run rails console it initiates an IRB. This means we don’t need to go through a web interface to update our app eg create articles. We can then type Article.new to create an empty article. It shows that we have the headings title, body and two timestamp headings

**LOOKING AT THE MODEL**

The code for the model is kept in app/models/article.rb. Currently this file is pretty empty with no attributes defined. Rails is really clever in that it already knows that an article should have a title and a body. It does this using a technique called reflection. Rails looks at the database, sees the column headings and assumes that these headings should be the attributes in the model. There will also be an attribute called id. Rails gives each entry a unique id so that we can refer to it if needed.

To make a new article in the console we type the below line by line

a = Article.new

a.title =”sample title”

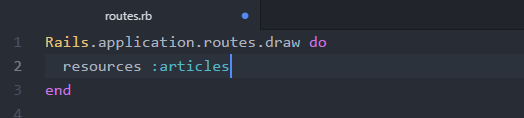
a.body = “sample text in the body”

a.save

This input will create our first article and save it! As well as filling in the title and body, rails will fill in the id and timestamps for me. The id was 1. I can view the article using Article.all

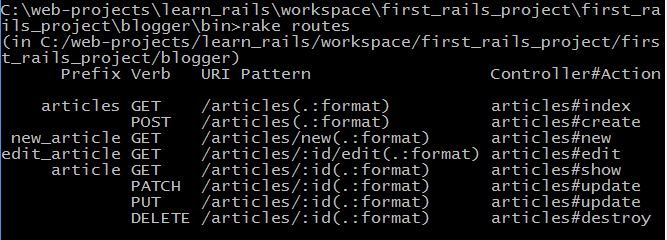
**SETTING UP A ROUTER**

Out of the MVC architecture we have made the model. Now we need to create the view and the controller. When a browser speaks to a rails app the first thing rails does is go to the router. The router is able to decided what the request is trying to do and what it needs. It figures it out using the address it is requesting and other http parameter such as Get or Put. We find the router file in confige/routes.rb. This file is pretty empty until I start building routes.



We edit the file to something like the above. I added resources :articles. What this block is doing is telling the router to expect requests which follow RESTful principles What this means is that when I get a request like “localhost3000/articles/” the router will understand that I want a list of articles. I don’t have to specifically tell it(?) Also it would know that “localhost3000/articles/new/” means I want to write a new article.

Line two, although simple will create all the typical routes I will need(in accordance with rails’ restfull principles). I can view them by doing rake routes and I get the below. If I delete line two my routes disappear!



Let’s look at the first row. Articles is the first prefix in the list. The router will provide two methods (index and create) using “articles”, the articles\_path and article\_url. The \_path version is relative while the \_url version uses a full url. The path version is always preferred.

The second column says “get”, this is the verb for the HTTP route. Web browsers typically send requests with the verbs get or post.

The third column- *I don’t understand it* but the part in brackets is optional. The uri pattern in the example will match articles/, articles, and articles.json.

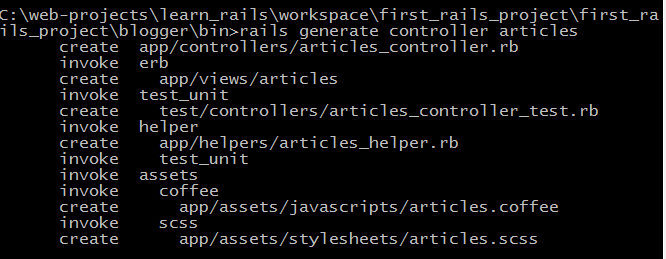
The fourth column shows where the route will map to in the application. In our example the route will map to the index method of the Articlescontroller class.

Now we have our router knows how to handle requests about articles we need a place to send the requests-the controller

(So the router has several routes to the controller. Which route it takes will depend on what the browser has asked.)

**Making a controller**

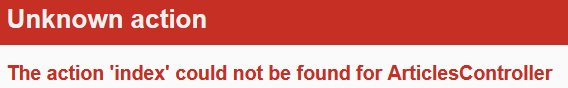
To generate a controller for the Articles we use rails generate controller articles. This make several files for us seen below.



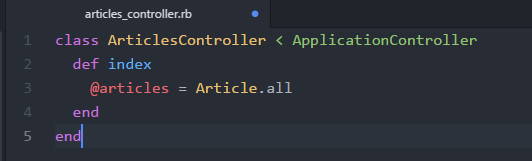
At the moment the first two creations are important. The first is the articles\_controller file (the actual controller) and the second is an empty directory called views/articles. This directory will hold the templates for my controllers view.

The first route we want to specify is for an index page. This should show an index of all of our articles. The index page will show when a user requests <http://localhost:3000/articles/>.

Rails uses restfull principles to assume that the above is true. When the router sees <http://localhost:3000/articles/> it knows to use the index action within the articles controller. We haven’t made this yet so if we go to the address we get the following error.

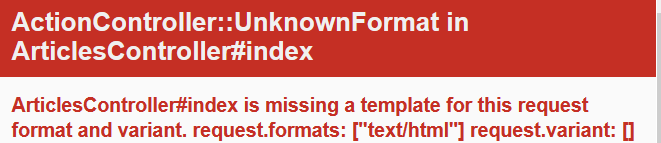


This proves that rails is working on restfull principles because I haven’t told it to look for the index action( it just knows!). So we add the index action to the articles controller



We have defined the action (method) “index”. The action has an instance variable called “@articles” and its value is all the articles. If “@articles” was not an instance variable it would only exist within the index method. Our rails app needs to be able to access the variable outside of this action. Rails need to use @articles in the view so it has to be an instance variable, so that it has a larger scope (not a local scope)

If we refresh our address we get a new error



This is telling us that the app is looking for a view template in the right place but couldn’t find one called index.html.erb. So rails automatically knows that our index action should have a corresponding index view template in the view folder!

CREATING A VIEW

In our empty views/articles directory I have named a file “index.html.erb”. The .html is usefull because it makes it clear that this template is for geneating html. Rails assumes that localhost3000/articles/ will be looking for a html page. I f we later create a file names index.xml.erb it will not show it for the articles request.

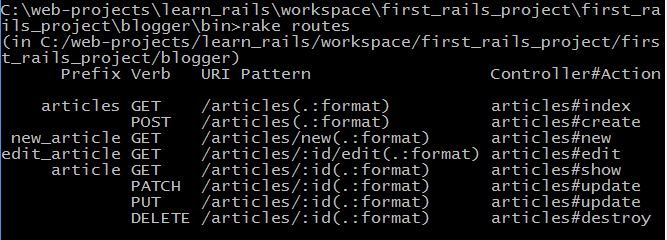
Within the file I have added the code below. It is a mixture of html and erb tags.



ERB is a templating language which allows usto put Ruby into our html template. An ERB clause starts with <% or <%= and ends with %>. If the clause starts with <% the result of the ruby code will be processed but not shown. If the clause starts with <%= the ruby code will be output in the place of the clause. This this syntax creates a header then an un ordered list. The list has an id of articles. Before the first item in the list we call a block on the instance variable @articles (remember this from the index action) This instance variable holds all our articles. The block iterates through the list assigning the variable article to each article. We then have our html tag for the list item. Below is an ERB clause which requests that the template shows the titles of the articles in our database. We then end the list and end the ERB block and the html un ordered list. As a result of this local3000/articles now shows the below in the browser!



CREATING LINKS

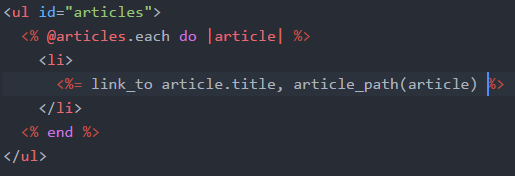


Above is the list of routes which are available to my articles. We can use the routes helper to create links in our app. If I want to see one article I need to use the 5th route. The address for this would look like localhost3000/article/id- Note that the article is singular and it needs the id of the article. In order to use this I will have to make a show action in my articles controller and then create a view for the show action 😊

In order to generate a url like “ /article/id” I need to use the route helper article\_path(id). For example article\_path(1) will generate the url string “/article/1”. We place the route helper within our html on the page that we want the link to appear on.

SIDE NOTE- Here we are using article\_path- This will generate a realative path to a file within our application eg app/articles/views/sample\_article If I used article\_url it would generate a web address instead eg localhost3000/articles/sample\_article. The path is preferred.

On our index view page we might want to make the title of each article a link to the full article. We can do this by amending the existing title ERB to this.



We have changed <%= article.title %> to <%= link\_to article.title, article\_path(article) %>

The first part is the link\_to helper. This tells ruby that the text that follows should be displayed and made into a link. In this case the article’s title will be displayed as a link. The next part is the route helper. This helper will generate the address of the page we want to visit upon clicking the article title. It generates html like the below.

<a href=”/articles/1”>First Sample Article</a>

Now I want to make a link at the bottom of the page which will take me to the page where I make new articles. To do this I have added the below to the index template.

<%= link\_to "Create a New Article", new\_article\_path %>

It is similar to the other link but the path I used is different. It uses the new article route which in turn uses the article#new action (which I haven’t made yet).

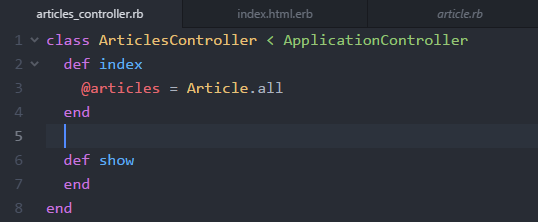
Lets not forget that so far my page is very boring. It looks terrible. We will want to be able to style this link with css later. To do this the link will need a HTML attribute/class. To add the attribute to the link we add it to the end of the link in the style of a ruby hash as below. We will now be able to style the link later on.

<% link\_to “Create a New Article”, new\_article\_path, class: ‘article\_title’ %>

**THE SHOW ACTION**

In order for our article links to work we will need to define the show action in the Articles\_controller and then create a view to see them.

We can define the show action under the index method as below.



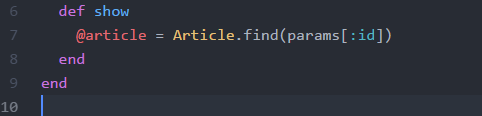
Remember that actions are just the name for methods which exist within a controller. So the articles\_controller’s actions are its methods.

Now if we click the first article title our error message changes to “missing template”.

SIDE NOTE

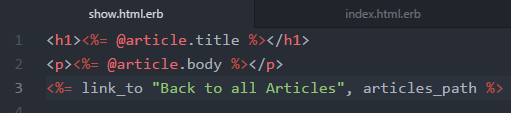
*When we click the first article, the URL is* <http://localhost:3000/articles/1> *. This URL was generated using the article\_path(article) route. It found the id of the article in the database and added it to the end of the URL. When we click the article we want to be able to see the title and body of the article. We are able to use the number at the end of the URL to search through the database if we use params.*

*Params is a method within the controller which will return a hash of the requested parameters/value. We often call the params method the params hash (even though it’s a method which return a hash!) Within the hash we can access the article ID. We add the below to the show action.*



*This creates a new instance variable @article (singular). Show will now look at the class Article and find the id parameter and return is as the value of a hash.*

In order for the show action to show the article we need to create a view. The view is called show.html.erb and contains the code below.



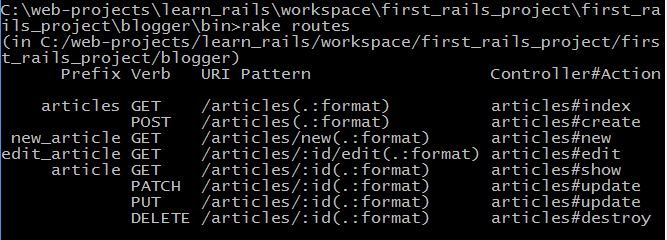
The code uses the instance variable I made in the show action under the articles controllers class. The page will display the articles title and the articles text. It also has a link at the bottom of the page which takes you back to the list of articles.

\*\*At this point, the tutorial has told me to download a css stylesheet to make my app more funky. I have C/P the css into app/assets/stylesheets/blogger.css. Rails know how to find the stylesheet automatically.

CREATING A FORM

So far I have added articles to my app using the console and command line. Most users wont want this so I need to make a form to accept article entries and that can save them to the database.

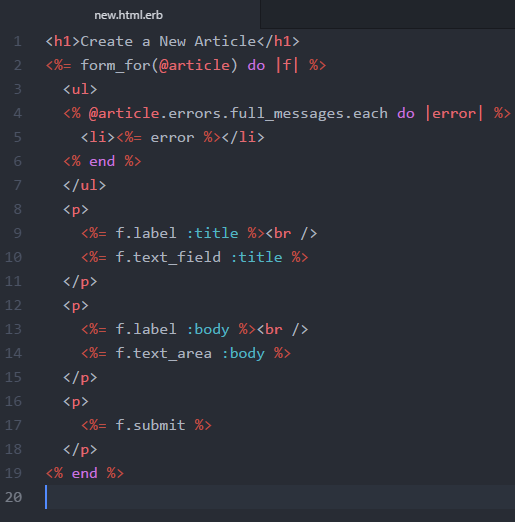
In order to create a new article we need to look at our routes.



We will be using new\_article and the controller/ method for this needs to be defined and a view created. I already have a link to my new view on the index page. I made it earlier using new\_article\_path. This path generates the URL <http://localhost:3000/articles/new>.

I have defined the method in the controller and created a view with the heading “create a new article”. The file name for the view is new.html.erb. The file name is important to rails. New matches the name of the controller. HTML tells rails to output the file as HTML to the user and erb is the language the file is written in.

We also now need to add a form to the new page.



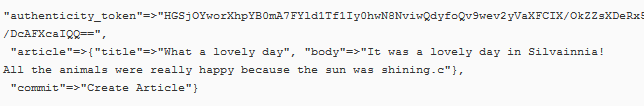
Building forms in rails is easy because it provides us with helpers. The first helper is Forms\_for(). It accepts one parameter and a block. The first line says “Please make a form for the @articles instance variable and name all parts of the form ‘f’”. The next helper is .label. .label makes a HTML tag for the field which will help us later in development. The f.text\_field is another helper which will make a single line textbox. We have given this text box the title :title. F.text\_area is another helper which makes a textbox but this one will have multiple lines and the name :body. The last helper is .submit- This gives us a button which will be called create.

At this point the form wont work and it is because of the @article object. When we run the form we pass @article to form\_for but we haven’t made and article out of @article yet! Because of this, we are effectively passing nill to the form and get nill in reply. In order to fix this, we need to go to the article controller, to the new action and add @article= Article.new. So now when we pass form\_for @article we are passing it a new instance of the Article class which the form can fill in. Now our form can accept data!



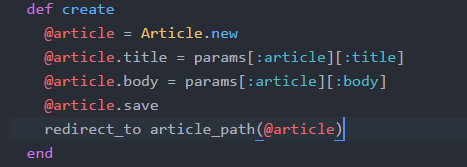
Create Button

In order to make the create button work we need to use the create action in our routes. We define create in our articles controller’s file. The form still does not work. If we add fail to the create action the browser gives us this.



We can see that there is an authenticity token for security followed by a nested hash. Articles points to a nested with the data from the article- both the title and the body. The “commit” key holds the text on the button the user clicked. (the server sees all buttons the same apart from the value of the commit key. So the fail method is useful because it allows us to see the structure of the data we want to use. We want to see the parameters for article, title and body.

In the create action we add this



This mimics what we were doing in the console to create new articles.

a = Article.new

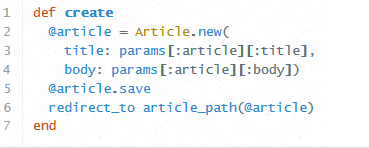
a.title =”sample title”

a.body = “sample text in the body”

a.save

It creates and object of new article. Then we use the params method. Remember the paramas method will give us a hash of the data that we want to find. Just like when we wanted to find the id earlier. We firt need to find the :article symbol and then find the :title/:body symbol which is inside it. The params method will then retrn the value of these symbols in a hash. The final line uses the redirect helper. This will change our location for us. In this case it will take us to VIEW PAGE OF THE ARTICLE WE JUST CREATED.

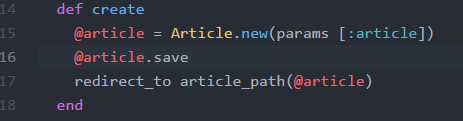
Although what we have done works, it is not best practice to fill the controller up with this much code. The controller is a middle man and should know as little as possible.

We could change our code to look like this 

In this instance we have called new on Article and given it a hash of the attributes. This will also work but it a bit silly. We are using params to find hashes and then creating a new hash of the same information. Instead the simplest thing to do is this.

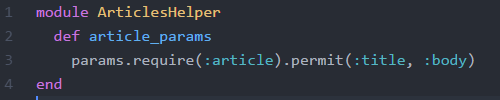


So now create will make a new article and give it the values of the keys in the nested articles hash. Our create method now looks like this.



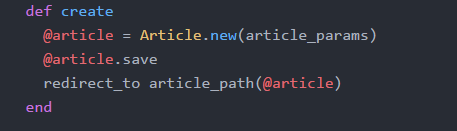
Unfortunately rails ‘ security stops this method from working. It is because it’s not a good idea to blindly save parameters which are sent to us via the params hash (im not sure why not.

So we need to make rails feel more secure. We do that by creating strong parameters.



The code above has been defined in the helpers directory in the article\_helpers.rb file. The method uses the require and permit methods to make sure that only these parameters can be saved into the model. To use this module we must type “include ArticlesHelper” at the top of our controllers list.

Because we have this helper we can now make our create controller even shorter by getting rid of params and replacing it with a reference to the article\_params method.



(this is all getting kind of confusing but I think I understand it. It s just al lot to keep in ones head at once.

**DELETING ARTICLES**

First we make our delete button. We could have it on the index page but I want it on the show page. To make it we can use <%= link\_to “Delete Article”, article\_path(@article) %>. This creates us a button which will show the text “Delete Article”. If we look at the article routes table we will be using the Destory controller which uses the article\_path with an id.

When we click the button in the browser our server comes up with the following error

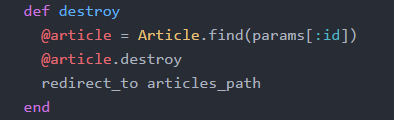


This part of the error is surprising. Articles/1 is perfect for the path but the verb is wrong. It says get instead of delete. This is because in HTML we can only use the verbs GET and POST. In order for rails to work around this we use some javascript magic to make a fake delete verb. Rails can now pretend that clicking this button creates the delete verb.



Now when we click the button we get our usual error saying that the destroy action is not in the articles controller.

We add the below action to the articles controller. This action will find the article we are viewing by id and then destroy it. When it is destroyed we will be redirected o the articles index. 😊

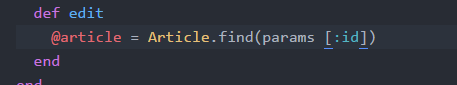


Because deleting is dangerous it is wise to add a confirmation request when our button is clicked. We add “data: {confirm: “Are you sure you want to delete this article?”} to our delete button in the view. This will give us a js pop up. Clicking yes will delete and clicking cancel will stop the request.

**EDITING**

Just as create and new are a pair, edit is paired with update. I want create an edit button on my show view. This looks like this <%= link\_to “Edit Article”, edit\_article\_path(@article) %>. The button will say edit article and the path will allow this artle to be edited. Now we need to define the controller.

All the edit action does is find the article and display the form

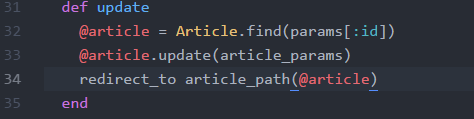


\*note the space after params broke my app so it was removed. We now need to create a view for the edit action.

The form that we are going to use is identical to the form we used to create article. If we remember the dry principles it goes against Ruby principles to have two pages showing the same thing. What we can do instead is create a partial file to hold the form and link to it from both the new and the edit views. Partials are used when we create reusable code which we want to render on several different views. Files names for partials always start with an underscore. We save our form partial in app/views/article/\_form.html.erb (with all our view files)

To the \_form.html.erb we add all the code that creates the form and save the file. Now we can delete the form from the new.html.erb view and replace it with <%= render partial: ‘form’ %>. We do the same thing to our edit view and we have our form displayed and two tiny view files. I believe that the submit button on my form automatically know to call itself update article due to rails magic. In the create view it calls the button create article! Very very clever!

Now it is time to use the UPDATE route



I have defined update in my controllers. The first line finds the article by id from the database. The second line uses the .update method to amend the article. It access the article by using the (article\_params) method which we coded earlier in the articles helper file. We are now able to update our articles.

The final line is a redirect to the edited article. The update controller is very similar to the create controller except .update saves the changes automatically so we don’t have to type @article.save in the update controller

ADDING A FLASH!

Flash notices are a form of feedback for the user. It confirms that the action they commanded was carried out for them. It’s a good idea to add a flash to the controller for update in the manner below.

flash.notice = “Article ‘#{@article.title}’ was updated”

But we don’t see it yet when we update an article. We need to add it to a view. The update controller redirects to the show view so it would make sense to add it to this page.BUT WE WONT! Because we want to use the flash on many pages we should add it to the “layout”. The Layout is found in our views directory under the name application.html.erb. This file holds the application layout. It is used to wrap multiple view templates in our application although we can make layouts for each individual controller. It is basically the standard stuff we have on every view- I imagine we could put our header and footer in here and then they would appear in every view. If we look at the layout we see <%= yield %> this is the syntax which displays our views. We should put the flash just above the yield.

In HTML it looks like this <p class “flash”><%= flash.notice %></p>

So now when I update an article a paragraph pops up at the top of the page (above the view) that says the article was updated.

THE SITE ROOT

The site root is the first page of the site. Our homepage is <http://localhost:3000/> and currently this is still showing the rails homepage by default. We can change this if we go to our routes and add another at the top. We type root to: ‘article#index’ and as if by magic rails knows to follow the index route to the index view.

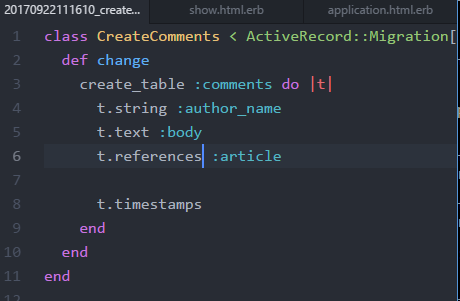
ADDING COMMENTS

After we post an article we want other users to be able to comment on them. We will need to store these comments in the database so we can start by creating a comments model. The model will need to hold the name to the author, the comment and an id linking it to the correct article.

When we made the article model we made it empty and then added the fields later in our text editor. This time we will generate the fields in the commandline at the same time as the model.

/rails generate model Comment author\_name:string body:text article:references

This command generated this migration



Because the migration already has the fields we need we can run the migration. Using the command rake db:migrate-this runs all unrun migrations. We have now created a table called comments in our database.

SQL is very powerfull because it lets us show how data in different tables is related. For example it will let us connect one article to many comments. We can connect these two tables using foreign keys. Foreign keys are away to mark one to one and one to many relationships. In our case it will be to connect one article to all its comments.

Rails makes working with these relationships very easy because it will make foreign keys for me. When I made the migration for comments I named a references field “article”. Rails knows that when I typed “article” I was referring to the article table in the database. Rails will therefore create the column article\_id using the following conventions.

The “many” objects should have a foreign key referencing the solo object

The foreign key should be titled with the name of the solo object followed by an underscore and id.

So our foreign key will be stored on the comments table and be called “article\_id”.

We can now go to our model files (where the comment/article classes are defined and define the relationship of the classes. In comment.rb we add “belongs\_to :article. In article.rb we add “has\_many :comments. This explains to rails, the one to many relationship.

We can add comments to articles using the command line as below

a = Article.first- assigns the first article to “a”

a.comments- calling the comments method on “a” gives a blank array because “a” has no comments, yet.

Comment.new- creates a blank comment object with fields that say nill

a.comments.new- this is what matters, this shows “a” with its comments. The comment is blank except the id for the article of “a”

a.comments- shows all the comments attached to a- in our case it is only one.

We can add a second comment to “a” with content in two ways

c = a.comments.new

c.author\_name = “daffy duck”

c.body = “This is daffy’s comment”

c.save

OR

d = a.comments.create(author\_name: “Gregory”, body: “I love this article”)

The first method uses new- which requires the .save method to commit it to the database whereas .create will save automatically- just like in my controllers for articles.

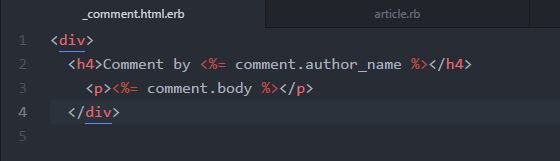
We now want to be able to view our comments under the article which they belong to. We will do this by creating a partial and putting a link to it on the show page.

<h3>Comments</h3>

<%= render partial: ‘articles/comment’, collection: @article.comments %>

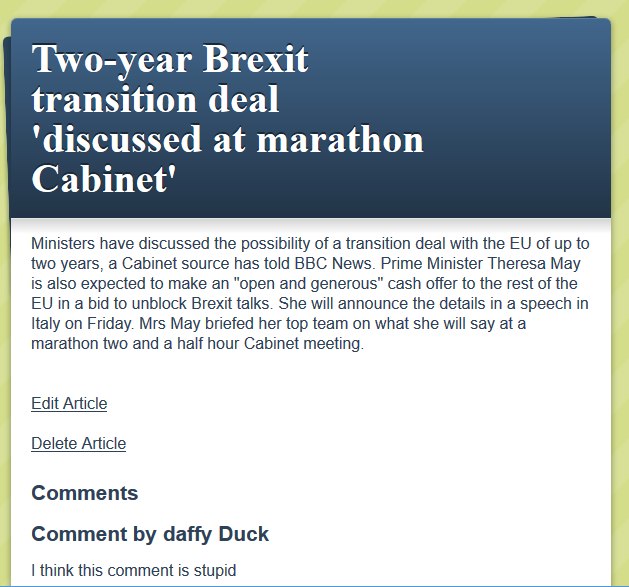
This will render a partial named “comment” (stored within article). The “collection” says that we want to render the partial once for each object in the collection of @article.comments (basically we will render all comments for the article).

Now we will make the partial called /\_comment.html.erb



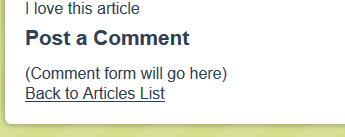
We create a div with the header Comment by author\_name with a paragraph below showing the comment.

I can now view my comments under the article



We now want our user to be able to add comment to the article when displayed. We need a comment form to appear below the article. We will make partial for the form and render it onto the page. The partial will be saved under apps/views/comments/\_form.html.erb- I had to create the comments directory. Now I can render this to the show view using the code below.

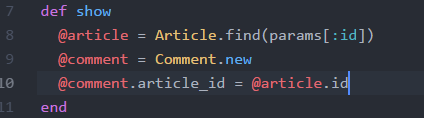
<%= render partial: ‘comments/form’ %>



A bit of html on the form partial shows that my furture form will be displayed.

**Creating a Form**

Before we make our form we need to make rails aware of the field we have in our comments table. When we wanted a form to create a new article we had to create a blank article in the new controller.We did this so that rails was aware of the fields the article has. The form was then generated on the new view. We want our comments form on the show view so we need to implement a new comment under the show controller.



Line 9 creates a new instance of the comment class (a blank object with the parameters of author name , body and id)

Line 10 manually assigns the article\_id attribute to the id of the article.(something to do with rails’ mass assignment)

If we go ahead and create the form using the below code we get the following error

<h3>Post a Comment</h3>

<%= form\_for [ @article, @comment ] do |f| %>

<p>

<%= f.label :author\_name %><br/>

<%= f.text\_field :author\_name %>

</p>

<p>

<%= f.label :body %><br/>

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

</p>

<p>

<%= f.submit 'Submit' %>

</p>

<% end %>

NoMethodError in Articles#show

Showing app/views/comments/\_form.html.erb where line #3 raised:

undefined method `article\_comments\_path' for #<ActionView::Base:0x10446e510>

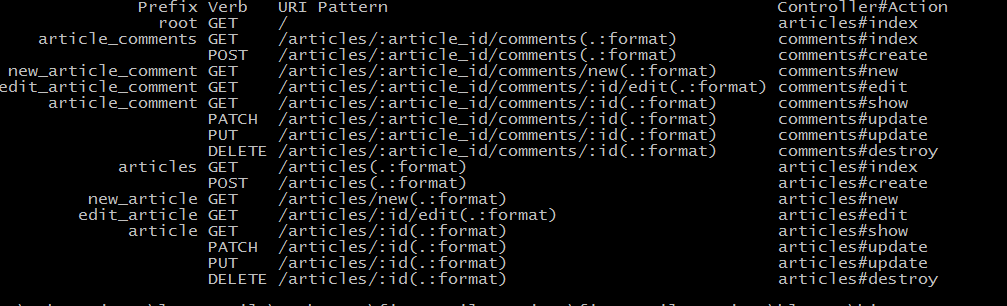
The form helper is trying to submit the form using a path called article\_comments\_path. That’s a helper we expect to be made by the router file . We need to tell the router about “Comments”. We can add “Comments as a sub resource to articles to create this route. Like below

resources :articles do

resources :comments

end

Our form is now displayed and we have made all of the below new routes. We use the comments#create route.



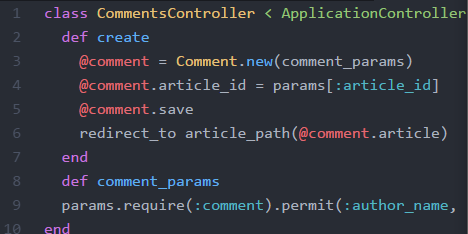
Now I can see my form. If I try to submit it however I get the following error.



This is because I have an article\_ontroller, but not a comments controller. (Even though my comments routes are sub routes to my article routes I still need a comments controller to process them) I can create the comments controller in the command line by typing

rails generate controller comments

We can then create a new comment using the code below



This code will make our comment and also redirect us to the article and comments below it when we are done. When we use the redirect we can’t redirect to @article. This is because @article isn’t defined within the comments controller. However @comment.article gives the same result because .article represents article\_id

**Adding a comments counter.**

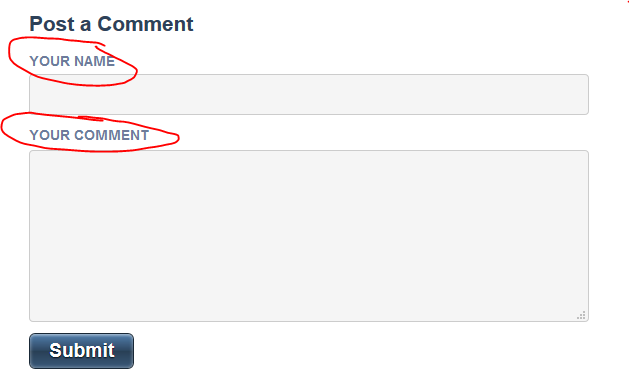
This is really easy. If we want the header above our comments to say he number of comments below the header we can amend the comments header on the show page to this.

<h3>Comments (<%= @article.comments.size %>)</h3>. This gives this result



We can also make some edits to the comment form. It currently says Author name and body above the fields. These are automatically generated due to assumed defaults. We can change them if we give rails strings for them as below

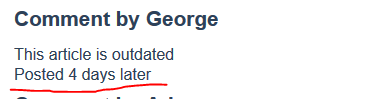




We can also add a timestamp to the comment display. Rails has a great helper which will give us a written version of the time which has passed since the comment was posted. We can add the code below to the \_comment.html.erb

<p>Posted <%= distance\_of\_time\_in\_words(comment.article.created\_at, comment.created\_at) %> later </p>

This code works out the difference between when the article was posted and when the comment was posted and returns a string



**Tagging**

Tags are used on blogs to assign different article to different categories. Tagging aids organisation and navigation. I might tag my Brexit article with “Brexit”, “Europe”, “uk”, “Theresa may”. This would allow users who click on one of these tags to find all articles relating to the tag.

An article can have many tags and the same tag can be used on many articles so the relationship is described as “many to many”. It doesn’t make sense to describe many-to-many relationships in the same way as we described one -to-many (eg using the id of one as the foreign id of the many).

One solution would be to make a “join table” This would list all the tags and the ids of each article that they belong to. Traditionally we would call the table article\_tags. Rails would describe the relationship in the Article model as “has\_and\_belongs\_to\_many” tags and but the samein the Tags model.

However there is a better solution. We can promote relationship between tags and articles to be its own model named “tagging”. It will show the relationship between articles and tags.

An article has many taggings

A tag has many taggings

A tagging belongs to an Article and belongs to a tag

So we need to generate a model for tags and a model for taggings

Rails generate model Tag name:string

Rails generate model Tagging tag:references article:references

Rake db:migrate

The tags table has a column names “name” and the value is a string

The taggins table has two columns. The first is tag column with a value of reference. This is the tag\_id and hold the foreign key of the referenced tag. The article\_id hold the foreign key of the referenced article.

**Expressing Relationships**

Now we have our models we need to tell rails what the relationship is between them. In app/models/article.rb we add “has\_many :tagging”s. In app/models/tag.rb we add “ has\_many :taggings”

Our taggings model already has lines saying it belongs to tags and articles! Magic!

This is a comment type of relationship to have in rails. Unfortunately if we want to find the tags that belong to an article the syntax is long and complex see below.

Tags = article.taggings.collent{|taggings| tagging.tag}

This is not an elegant way to do things. Instead we can edit the models for article and and tag to make finding things simpler.

If we add this to article.rb

has\_many :tags, through: :taggings

This basically says that an article has many tags but we need to go through taggings to find them. We add something similar to the tag model

has\_many :articles, trough: :taggings

Now if we want to find all the tags for an article we just type article.tags. If we want to know the articles a tag corresponds to we just type tag.articles.

We are now able to start making tags in the console.

A = Article.first-this will call our first article from our article model.

a.tags.create name: “tag1”- this will create (and save) a tag called tag1 into our tags table. It will be saved with its name, creation time and updated time. It also saves the tag in the taggings table. It saves the tag\_id, the article\_id and the created and updated times. The tag id is 1 and the article id was 11.

If we create another tag and then call a.tags we can see our tags printed out in the console.

Now we need to be able to create an article and add a list of tags, separated by commas to the article. The rails should associate the tags with the article and create the tags if necessary. I have amended by \_form.html.erb to include the below. This will accept tags (eventually)

<p>

<%= f.label :tag\_list %> <br />

<%= f.text\_field :tag\_list %>

</p>

This adds a label to our form and below it a text box which accepts tags.

Now we get an error saying that there is no method called tag\_list. We need to define the method in the article.rb file.

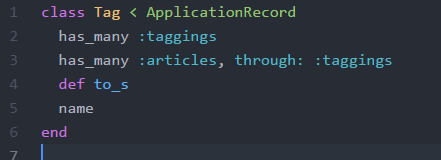


We can now see our tag list and box at the bottom of the new article form.

Our tag\_list method joins our tags together using a comma and by default prints them as a string. If we use the method in the console we get a strange output



The method gives an array of tag instances- not there names. The error is caused by the .join method calling to\_string. We can redfine the to\_string method in the tag class so that we can get the answer as the tag name.



But when we try to add tags to our article in the browser we get the following error.

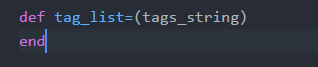


This is telling us that the parameter :tag\_list is not allowed in in our database because strong parameters is halting it. We have to add it to our permitted paramters in article\_params in the article\_helpers.

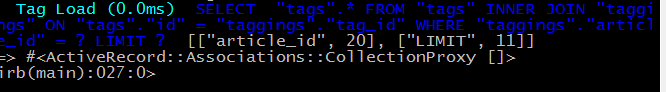
Now we get a new error



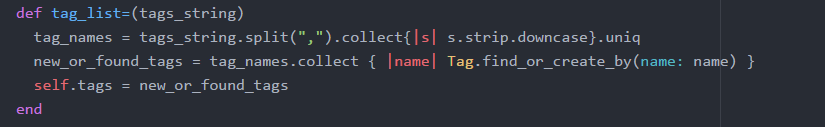
I don’t really understand anything from this point. But I have added the following code inside article.rb



Surprisingly this does fuck all to fix my problems. When I try to view the tags in the irb console I get an error!



My code didn’t make any tags. It gave me a blank []. We need to improve our tag\_list method.



This method is able to show me a list of the tag names. But we need to add code to the article.show page to display them. This si the code I used for this.

<p>

Tags:

<% @article.tags.each do |tag| %>

<%= link\_to tag.name, tag\_path(tag) %>

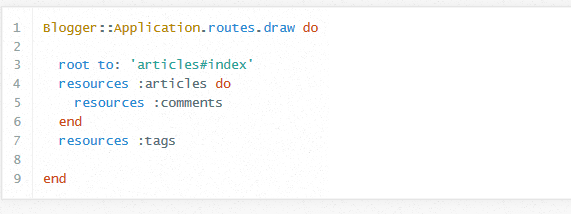
<% end %>

</p>

This produced an error looking for the tag\_path-Guess what! I need to make a controller and routes for tags

|  |  |
| --- | --- |
|  | bin/rails generate controller tags |

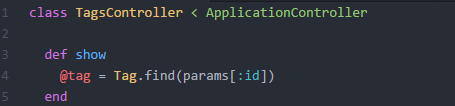
And



Now I am able to view tha tags associated with that article

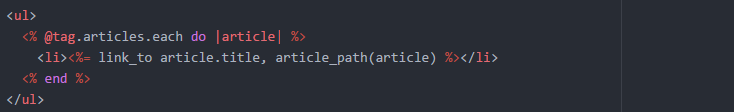
**Listing all articles by tag**

I now want to be able to click a tag and see a list of all the articles associated with that tag. Currently the links give the following error- no show defined! So I need to define show in my tags controller



This makes an object called @tag. The code after it find the id of the tag I just clicked and store it in @tag.

If we add the below code to our tag.show view we are able to see all the article which belong to that tag.



This code will create a link out of the article name and take me to the show of the article when clicked.

**Listing all tags**

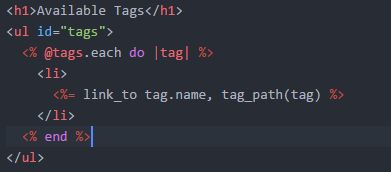
To list all tags I had to create a view for index in which I would show a long list of all of the current tags.

I defined the index action as below



This assings all tags to @tag.

In the index view I then added the below

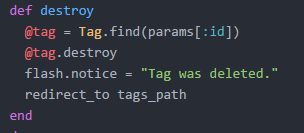


This take @tag and iterates through it, assigning tag to each instance. I can then print the name of each tag. The names become links to the tags show (which displays the articles associated with the tag).

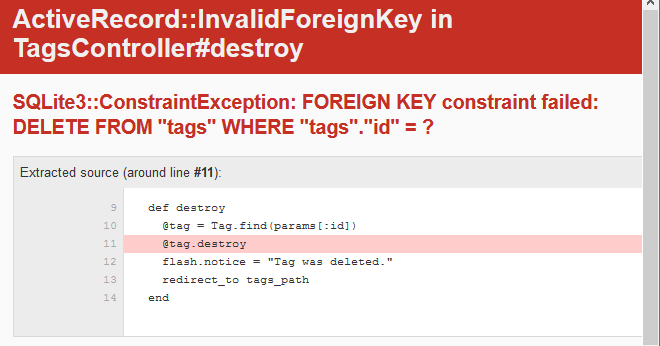
I now need to be able to delete a tag. I want to be able to delete a tag from the tag show so I added the below to the tag show



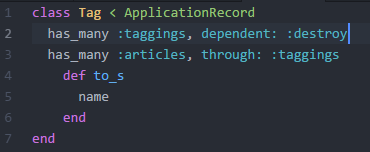
And then defined the destroy action in the tag controller as below



I copied these from the articles destroy action we made earlier- making only minor changes. However it would not work, I kept getting the following error

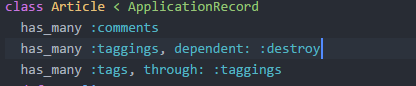


I found this very confusing because I don’t think it really makes sense. I found an answer online. They explained that there is a foreign key relationship between tags and something else in my app. If rails allowed me to delete the tag I would end up with orphaned article ids in my taggings model, cluttering it up and being useless. I need to be able to delete a tag and at the same time delete all associated taggings. To do this I edit my tag.rb file to add “dependent: :destroy”



<https://stackoverflow.com/questions/45007420/invalid-foreign-key-when-deleting-tags-in-rails>

If I make a new article and give it tags I can no longer delete it. To fix this I also had to add dependent destroy to the articles controller



Once the article is deleted the tag remains, stored in the tags model.

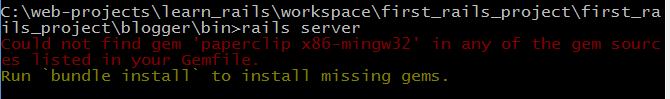
I am now able to delete tags and get on with my life!

**A few Gems**

The first gem we are going to add to our app is called paperclip. It will manage file attachments and uploading. In order to add the gem we need to go to our Gem file and add

gem “paperclip”- This line tells rails to load the paper clip gem when rails starts up.

Now when we reload our rails server we get the following error



Now now need to install the paperclip gem using bundle in the cmd line.

bundle

We get a lot of output and our paperclip gem is now installed.

Sidenote

paperclip is dependent on a program called imagemagick which can change the format of image files eg jpeg to png. I have also downloaded imagemagck.