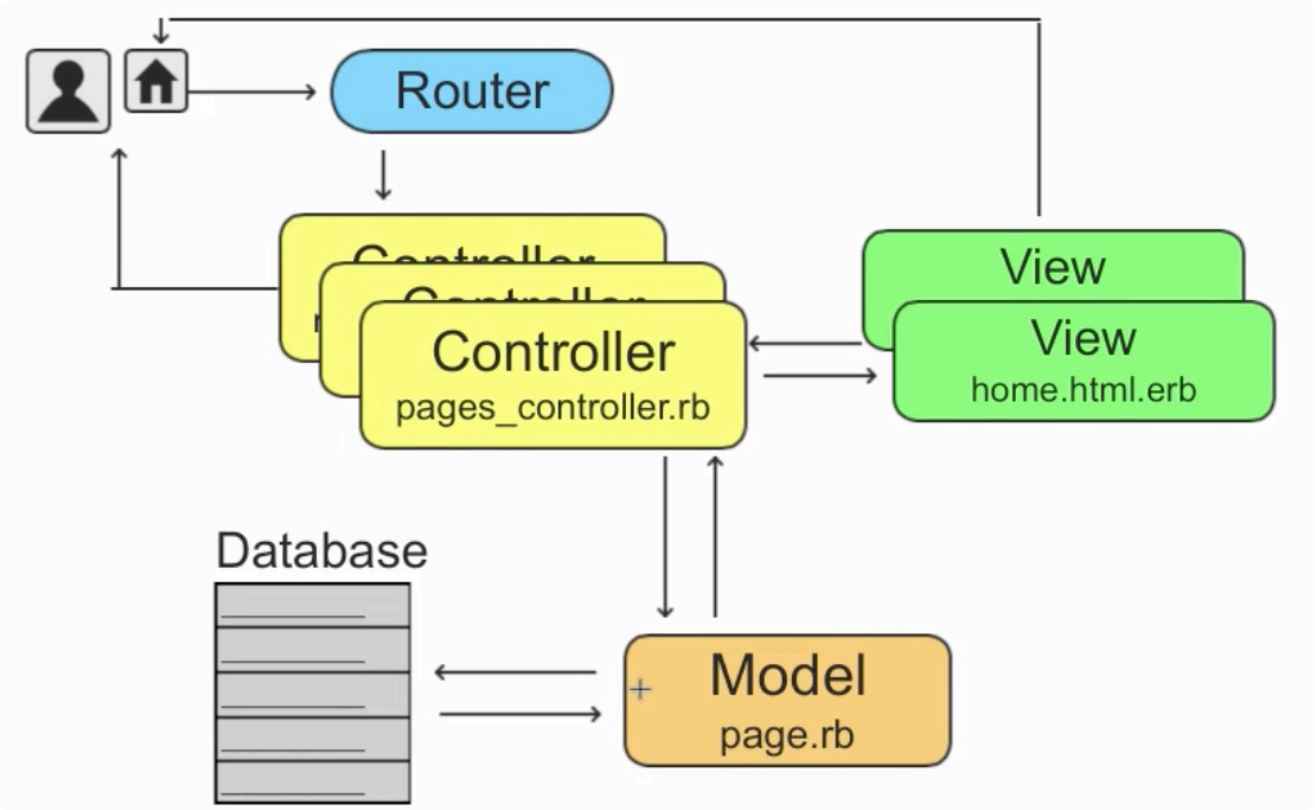
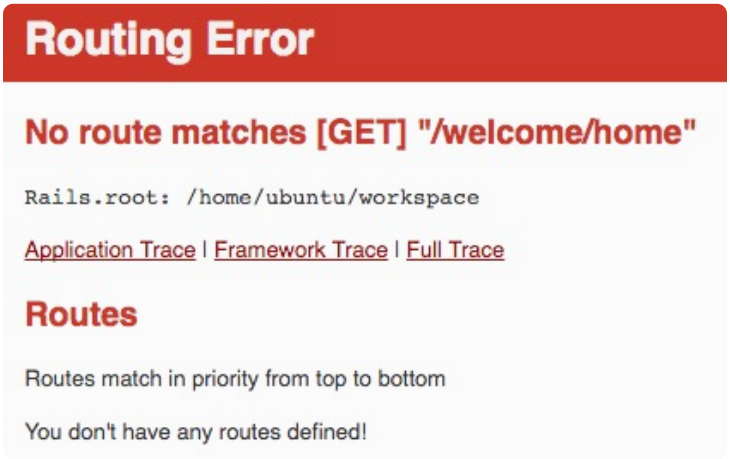
Basic Rails App Tutorial

After setting up a basic rails app with rails new:



The minimum needed for a rails app is a route, a controller, and a view. This creates a static site.

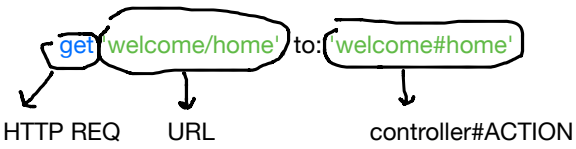
With the server running on the basic app, check the URL /welcome/home. Notice the error:



You can check the available routes with rake routes. As of now the basic app has none:

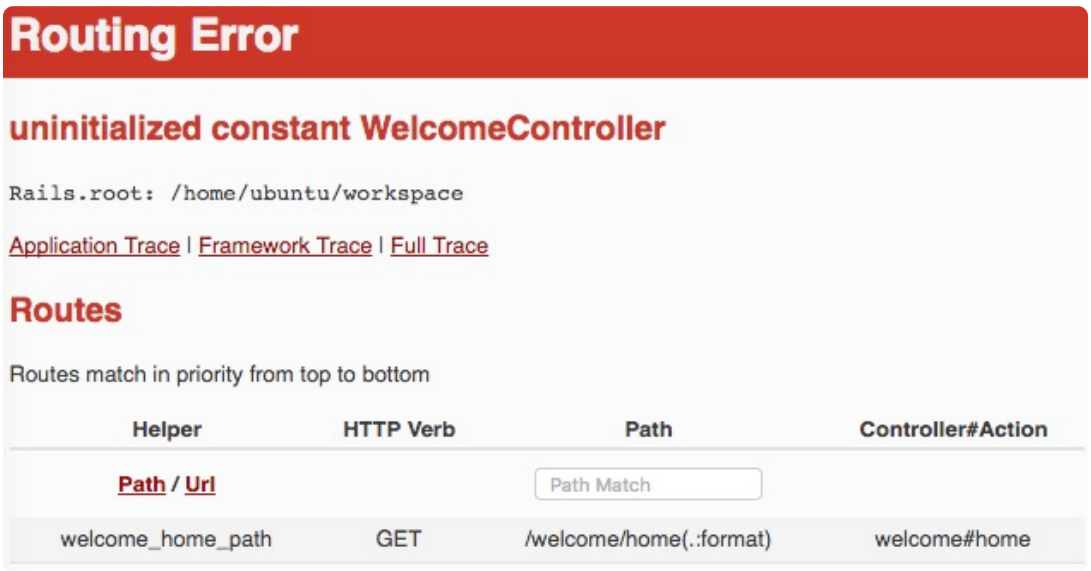
```
b_random_sb:~/workspace $ rake routes
You don't have any routes defined!

Please add some routes in config/routes.rb.

For more information about routes, see the Rails guide: http://guides.rubyonrails.org/routing.html.
```

When we add:




```
b_random_sb:~/workspace $ rake routes
      Prefix Verb URI Pattern                Controller#Action
welcome_home GET  /welcome/home(.:format) welcome#home
```

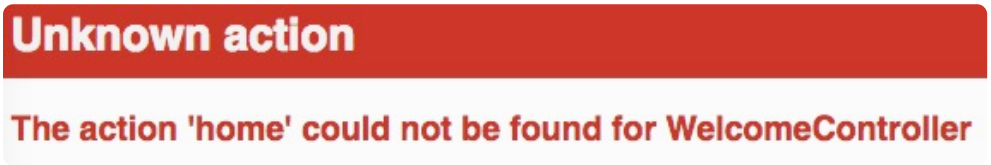
Now check the URL /welcome/home. A new error says we are missing the controller:



Now manually create welcome_controller.rb under app/controllers, and add the class.

```
class WelcomeController < ApplicationController
  |
end
```

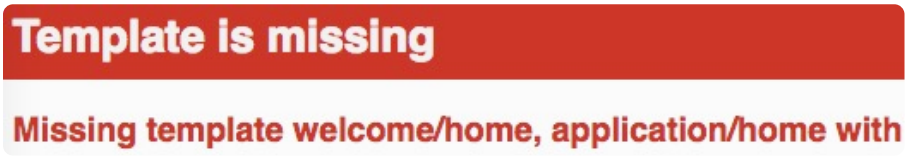
Run the URL again...



We have the right controller, but no home action.

```
class WelcomeController < ApplicationController
  def home

  end
end
```

Run the URL again, and:



This is saying that it found WelcomeController and the home action within, but there is no template, or view, specified. A controller should always have a corresponding view- *i.e., **welcome controller => welcome view***

Create /view/welcome and add the file ***home.html.erb***, and add some text to be rendered. Now the server should run successfully.

You could also call root 'application#home' in routes.rb, and then in application_controller.rb create:
This uses the default ***application.html.erb*** as the view

```
def home
  render html: 'This is home'
end
```

Add an about page the same way that we added a home page.

```
b_random_sb:~/workspace $ rake routes
       Prefix Verb URI Pattern                Controller#Action
 welcome_home GET  /welcome/home(.:format)    welcome#home
welcome_about GET  /welcome/about(.:format)   welcome#about
```

Then run rake routes to see the routes available to your app.

The info that rake routes shows is especially helpful when we want to link from one view to another- To go from the rendered 'about' page to the rendered 'home' page. From the rake routes output, we know that /welcome/home is the URL pattern, and that welcome#home => controller#action. But what about the first output, **Prefix? The prefix is ruby code- by adding "_path" to the end and using the rails **helper method(link_to)**, you can easily reference the URL pattern from within the app:

```
<%= link_to 'home', welcome_home_path %>
```

Adding this line with another view template creates a link to /welcome/home.

Now that we

1) Create a new rails app called alpha_blog

2) Ensure you can run the server and preview the app

3) Create a pages_controller.rb instead of welcome

4) Make sure routes work

5) Create a homepage and about page through pages_controller

6) Design the home and about page with HTML and CSS

At this point you should have a basic functioning static app

Picking up where we left off after building the new app, when we start the server up it still goes to the default Rails server page. This page is the root page. If we want the home page to be the root, we need to change it in routes to:

```
root 'pages#home'
```

But when we start the server our app gives an error:

NameError in Pages#home

If you read through the error, it has a problem with the helper pages_home_path, which we changed when we added the root. If we check rake routes, we see the new path is root_path. All the old pages_home_path's need to be updated to the new path, or Rails will keep giving an error. Update the paths and refresh the app- It should load without issue.

When we navigate to the about page, it still uses the controller as part of the URL address.

```
get 'pages/about', to: 'pages#about'
```

Remove the controller from the address and run rake routes to see the new helper path. Change out the old paths so all the links work.

```
get 'about', to: 'pages#about'
```

The URL's should be http://.....com or localhost:3000 for the homepage and add /about for the about page.

SET UP PRODUCTION ENVIRONMENT

We will be using Heroku for the production environment. Heroku does not does not support **sqlite**(db), so

postgresql(db) will be used. In the Gemfile, move **sqlite3** to the dev/test group(we still use it in the dev env.) and create a new group called *production*. Add the pg gem and the rails_12factor gem to production.

Run bundle install --without production. This will not load the production gems, but it does save the info to Gemfile.lock where the production environment can use it.

Commit to git

Now heroku create and enter login credentials. Heroku initiates a generically named site.

If this is the first time running heroku on the system, to avoid having to submit credentials every time run heroku keys:add

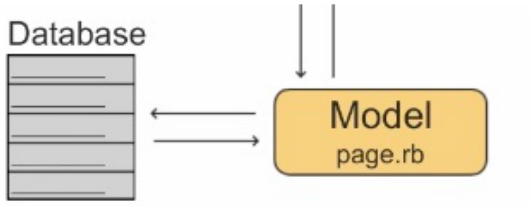
Commit to git

Then run git push heroku master.

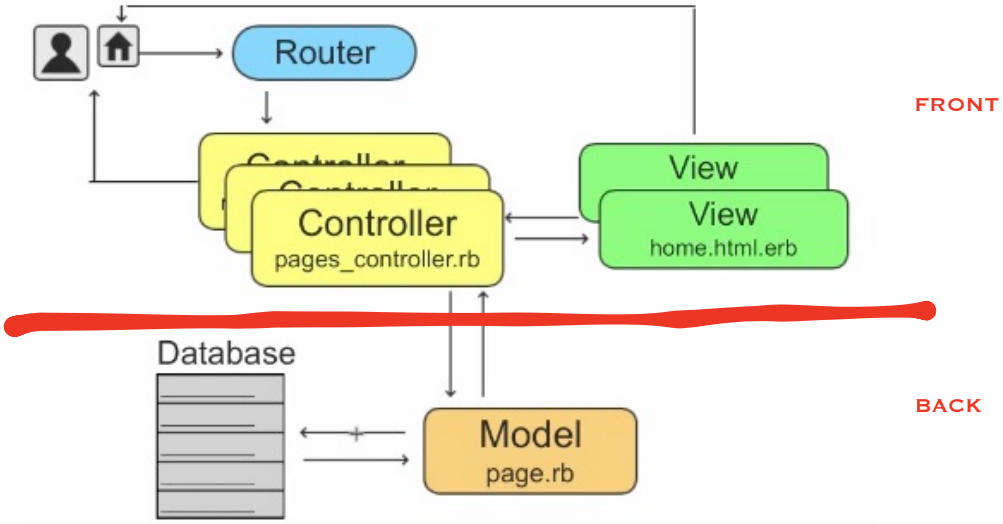
At this point heroku can run the static app.

To change the name from the default run heroku rename <new_name>. The new URL is <new_name>.herokuapp.com

Now lets move to the back end of the app- to the "M", or **model**, in MVC:



You can think of this separate from the rest of the MVC setup; the **router**, **controller** and **view**:



This is a db... think of it like a spreadsheet.

articles

ID (rails generated)	title	description	user_id
1	Great Weather	Great Weather outside today	3
2	Second Article	Today is a great day for coding	2
3	Rubyist day out	It's amazing what Rails can do!	6
4	4th Article	Rails is fun!	1
5	Article about Rails	This is a great article about rails	3

Class

Attributes

Each "article" has an ID, a title, a description and a user_ID(to show who created the instance of article)

To interact with the database, we use CRUD- **Create**, **Read**, **Update** and **Delete** an instance of *article*.

Usually in order to interact with the db, you would need to write SQL code. However, Rails uses Active Record, which means you can use ruby code to talk to the db. When the ruby runs, it gets converted to SQL.

This is where the **model** comes in. The **model** is what we use to interact with the db.

In the *test_app* lets generate some data in the db, a **model** to represent it, and all the other parts of an apps MVC. To do this, Rails has a magic command called scaffolding:

```
rails generate scaffold Article title:string description:text
```

scaffolding command

Class

attribute followed by data type ('text' is a for long strings, i.e. paragraphs)

Watch the terminal as scaffolding is being generated to see all the files being created. Everything you need is there... **Model**, **Views**, **Controller**.

The db data table will not be constructed until we migrate the info: `rake db:migrate`

Look at db/schema.rb to see what was built in the db with the migration, including a Rails timestamp. The *schema* keeps a

```ruby
ActiveRecord::Schema.define(version: 20160828042001) do

  create_table "articles", force: :cascade do |t|
    t.string   "title"
    t.text     "description"
    t.datetime "created_at",  null: false
    t.datetime "updated_at",  null: false
  end

end
```

running total of everything in the db:

This is directly related to the db/migrate file(s) created. If *schema* is a running total, the migration is an "each and only". Migrations modify the schema over time.

Check rake routes to see all the new routes. There are a lot more now. Under routes.rb, there is a new line:

```
resources :articles
```

****To see only the new article routes, run rake routes | grep article****

```
b_random_sb:~/workspace $ rake routes | grep articles
     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
```

in addition to our earlier routes:

```
 welcome_home GET    /welcome/home(.:format)      welcome#home
welcome_about GET    /welcome/about(.:format)     welcome#about
```

Now, when we add /articles to the end of the root URL, it will send us to the articles index page.

The index page has a link to create a new article instance. Clicking here take us to /articles/new URL which follows the new_article_path through **articles#new** (*articles controller#new function*)

Click and make a new article. When you go back to index, you see all the articles currently in the db, as well as, **Delete** and **Edit**

But where is this /new page and form coming from? **articles#new** == ArticlesController/ '**new**' method, which by default points to the articles views directory/ new.html.erb. This new view contains a _partial (more on partials later)

Once the 'new' form is filled out, you hit *submit*. *Submit* is the **Create** part of ***CRUD***. It follows the HTTP **POST** method under rake routes, which references **articles#create**. See the **create** action. Do not get confused by the CLI rake routes line:

```
new_article GET    /articles/:id(.:format)      articles#new
```

This line is for the helper method, URL and action of the 'new' view. Like the 'edit' view, these are **GET** requests. Once you **GET** the view, this is where the form is **POST**ed from.

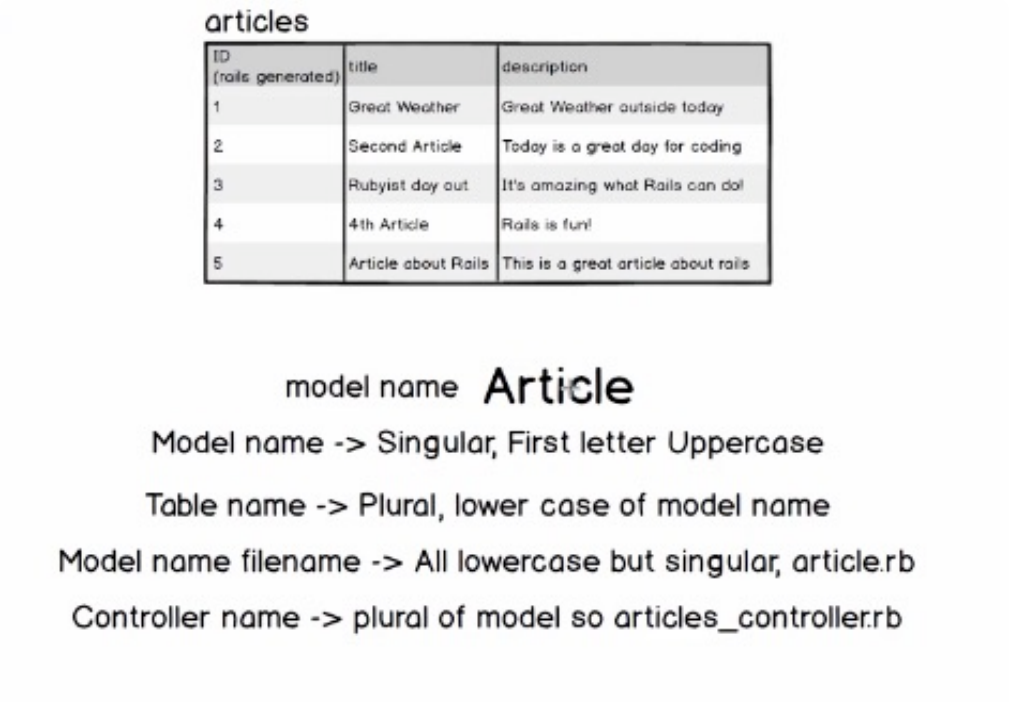
The same can be done when editing or deleting an Article instance. Submitting 'edit' follows the **PATCH** method and submitting 'delete' follows **DESTROY** (see rake routes). See the actions in ArticlesController.

All of this from a simple scaffolding command; index, new, edit, show, create, update & destroy

This would be a good time to learn a little more about SQL. It is necessary to interact with the db. Go to http://www.w3schools.com/sql/default.asp or see the SQL cheat sheet notes.

Models, Migrations and the Rails Console

The syntax conventions for Rails is shown here:



- With a model(or class) name **Article**, Rails by default will look for a corresponding table in the db called **articles**.
- The file that represents the model is **article.rb**.
- The controller is users_controller.rb.

This is the table we want to build in the alpha_blog.

To manually generate the MVC, without scaffolding, start by creating a migration:

```
rails generate migration create_articles
```

migration name

```ruby
class CreateArticles < ActiveRecord::Migration
  def change
    create_table :articles do |t|
    end
  end
end
```

This creates the empty **create_articles.rb**. See it at db/migrate. Inside, the migration uses a **change** method followed by a create_table method, to create **:articles** table. Rails is smart and picks out key words like 'create'

To build the table in the db, we need to run the migration- first, we need to add the attribute(s)

Now rub the migration, rake db:migrate.

```ruby
class CreateArticles < ActiveRecord::Migration
  def change
    create_table :articles do |t|
      t.string :title
    end
  end
end
```

Now that we have added data to the db, a schema.rb file is generated to show the state of the db at a given time.

But what if we forgot to add an attribute to the **:articles** table? What if we added the wrong attribute? To add another attribute there are a few options...

rake db:rollback will delete the table entirely. Run the rollback and check the schema- nothing is there. Its gone. The migration file is still there, though. Correct the migration file, then run rake db:migrate again and check the schema. Voila! Fixed it. This is not the preferred way to alter the db table... it works as long as another migration hasn't been run, but if a lot of data has been added to the table, it can cause a lot of confusion and wasted time.

A better way is to create a new migration file and add the new attribute(s):

```
rails generate migration add_description_to_articles
```

migration name

The new migration has an empty **change** method that we can use to update the **articles**.

```ruby
class AddDescriptionToArticles < ActiveRecord::Migration
  def change
    add_column :articles, :description, :text
  end
end
```

Here is the convention for adding the data: `helper_method :db_table, :new_attribute, :data_type`

Its also a good idea to add timestamps:

```
add_column :articles, :updated_at, :datetime
add_column :articles, :created_at, :datetime
```

Run the migration. It will only run the migrations that have not already been run. The schema, and therefor the **articles** table in the db, should look like:

```
ActiveRecord::Schema.define(version: 20160829183019) do

  create_table "articles", force: :cascade do |t|
    t.string   "title"
    t.text     "description"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

```
class Article < ActiveRecord::Base

end
```

Now that we have an articles table in the db, how do we communicate with the table/db?

If we reference the MVC diagram, we know we need a model to access the db. Now, in the /models directory create **article.rb**. This is where the **Article** class is represented, and directions for the Rails app to interact with the db table called **articles** go.

Once the model is created Rails whips up some magic and provides us with "*getters*" and "*setters*" for each attribute in the table that the model represents. Go to the rails console to test them out...

```
2.3.0 :001 > Article.all
  Article Load (4.2ms)  SELECT "articles".* FROM "articles"
 => #<ActiveRecord::Relation []>
```

Notice when we type *Article.all* to search the **articles** table, the SQL commands are queried. There aren't any articles yet, so an empty array is returned. When we run just *Article*, the console shows us each attribute and data type.

```
2.3.0 :004 > arts = Article.new
 => #<Article id: nil, title: nil, description: nil, created_at: nil, updat
ed_at: nil>
```

We can create a sample instance by entering *arts = Article.new*, which shows a value of nil for each attribute, including **id** since the instance was not actually saved. ****the *.new* method used here to generate the instance does not save by default.

Add a **title** with *arts.title = "add title here"*(make sure to use the correct data type). Now we can reference the new sample article we created- try calling it by name, *arts*:

```
2.3.0 :012 > arts
 => #<Article id: nil, title: "my first article", description: nil, created
_at: nil, updated_at: nil>
2.3.0 :013 >
```

It includes the new **title**. Add a **description** the same way... The rest of the attributes will be handled by Rails if we were to save this example as an actual Article instance- which we will do now with *arts.save*

```
2.3.0 :015 > arts.save
   (0.2ms)  begin transaction
  SQL (3.4ms)  INSERT INTO "articles" ("title", "description", "created_at"
, "updated_at") VALUES (?, ?, ?, ?)  [["title", "my first article"], ["desc
ription", "This is the description for arts"], ["created_at", "2016-08-30 0
3:48:09.361224"], ["updated_at", "2016-08-30 03:48:09.361224"]]
   (13.0ms)  commit transaction
 => true
```

Take note of the SQL code to insert the instance into **articles**. Now *Article.all* will show our new entry.

All the attributes can be assigned simultaneously with the "*setter*" method *article = Article.new(title: "This is the second article", description: "This is the second description")*. Now when *Article.all* is run two instances will show, but one hasn't been saved and doesn't include **id** and *timestamps*. Save the new data.

The previous "*setters*" used *.new*, followed by a separate save method. Using *.create* includes the save function and hits the db immediately: *articles = Article.create(title: "This is the third article", description: "This is the third description")*. No need to save.

What if we want to edit an **article**? First we want to find the instance with an **id** of 2, use *article = Article.find(2)*. Now the instance can be referenced easily with the variable "article". To change the title do *article.title = "the updated title..."*- its more overwriting, than it is deleting and replacing. Don't forget to save it, *article.save*.

The same can be used to delete an instance- use *article.destroy*. CAREFUL, no save method is needed.

What if we created an empty instance by running *article = Article.new* and then running *article.save* without naming the attributes? The empty instance will still hit the db as:

```
 => #<Article id: nil, title: nil, description: nil, created_at: nil, updat
ed_at: nil>
```

This is a problem. We must maintain "data integrity" in the db. But how do we prevent these instances? This is where validations come into play. Each model can have its own validations, or set of rules.

Here is a validation we add to ensure that each instance contains **title** data. For validations to work, Rails must be restarted. Or the command reload! from the console.

```
class Article < ActiveRecord::Base
  validates :title, presence: true
  |
end
```

```
2.3.0 :002 > article.save
   (0.2ms)  begin transaction
   (0.1ms)  rollback transaction
 => false
```

Try and add a new instance without a **title** and watch it get rejected with this message.

If you ever need to know why a validation rejected an instance, run *article(or variable).errors.full_message*.

Add a presence validation for **description** as well.

Now that our **articles** objects must have a **description** and **title**, what prevents adding single letter or number as valid data? Nothing... but we can add a validation that prevents single digit entries and limits the size of data:

```
class Article < ActiveRecord::Base
  validates :title, presence: true, length: { minimum: 3, maximum: 30 }
  validates :description, presence: true
end
```

Add a length validation for **description** as well.

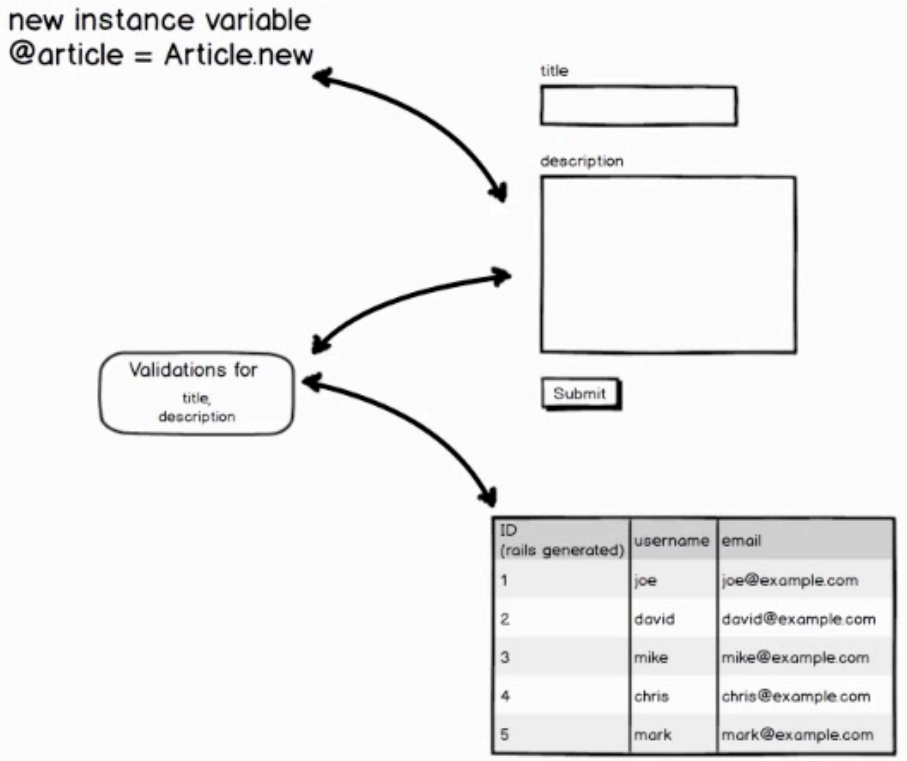
Example:

```
2.3.0 :020 > arts = Article.new(title: "do", description: "hello")
 => #<Article id: nil, title: "do", description: "hello", created_at: nil,
updated_at: nil>
2.3.0 :021 > arts.save
   (0.1ms)  begin transaction
   (0.1ms)  rollback transaction
 => false
2.3.0 :022 > arts.errors.full_messages
 => ["Title is too short (minimum is 3 characters)", "Description is too sh
ort (minimum is 10 characters)"]
```

Creating Instances from the Browser

In the test app we created an MVC for the **articles** table with scaffolding. This included everything we needed to create new articles i.e. controller, views, etc. In the alpha_blog we have created an **articles table,** and an **Article model** to communicate with the table. We also generated migrations and then looked at the schema to see the state of our table. At this point, however, we have no way to display or edit our table from the front end. No routes, no controller and no views...

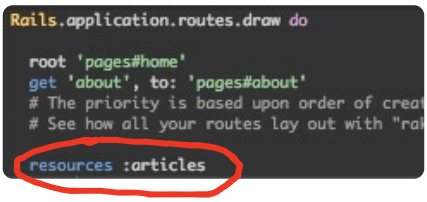
First lets take a look at the life cycle of these objects and how the object flows through the app:



As of now, rake routes shows what we have set- the root route(pages#home) and the /about(pages#about) page. When these routes are fed to the controller as actions, they default to the corresponding views.

We can begin to build the front end for the **articles** table by setting the basic HTTP RESTful routes. Rails makes this easy with the resources method. In routes.rb, set:

```
resources :table_name
```



This one line provides you with all these routes:

```
    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
```

Lets use /articles/new as our target, so we can create new objects...

Of course, if we try to navigate to /articles/new we get an error. There is no controller to relay the route.

We need a controller with a **new** action and a /view/articles/new.html.erb file. The **new** action defaults to the **new**.html.erb in views.

Now we just need to add the *form* that lets us input a new **articles** object to new.html.erb. Because this form will interact with the model, it is called a "model back form". Rails provides a convenient helper method, *form_for*, for creating these forms. To learn more about *form methods*, go to guides.rubyonrails.org/form_helpers.html

To create this form, do:

```
<%= form_for @article do |f| %>

<% end %>
```

If you run it, Rails is going to return an error since it doesn't know where the @article variable came from. Rails can't find it in the controller action. The **new** action is where we set @article to equal a new **articles** object, just like we did in the console.

The page will load now, but we still haven't finished the form:

```
<%= form_for @article do |f| %>
    <p>
        <%= f.label :title %>
        <%= f.text_field :title %>
    </p>
    <p>
        <%= f.label :description %>
        <%= f.text_aera :description %>
    </p>
    <p>
        <%= f.submit %>
    </p>
<% end %>
```

<==>

The form is complete. When you fill out the form and submit it, you are sending the **POST** request to the server (including the data from the form), then the application receives that data and performs some action with it, like creating a db record.

```
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"XKlxFOQaQpP2wQ7NR9/TxwyC
SxIXKlHnwt33F501MC779AUxVz+tAgBi5ZMzD1klcfC6cxxkzBzKUwJMylAW3g==", "article
"=>{"title"=>"xxxxxxxxx", "description"=>"desc xxxxxxxxxx"}, "commit"=>"Cr
eate Article"}
```

The params include some security info- We want to utilize the **CREATE** action, but first we want to specify what parts of the params hash we want the **CREATE** action to receive. For security reasons we put those specifications in a private action.

This method is storing the *article params*, but only the **title** and **description**.

```
#private
def article_params
  params.require(:article).permit(:title, :description)
end
```

Now the **CREATE** action calls the saved params method and saves them to the db- just like we did in the console.

```
def create
  @article = Article.new(article_params)
  @article.save
end
```

At this point, submitting the form to create a new object instance displays an error. However, if you check *Article.all* in the console you will notice the new object did get saved to the db. This error is flagged because there is no template and the **CREATE** action doesn't know what to do after saving. If we redirect to the objects **SHOW** view we have a place to land and the added visual proof of the object just created, add:

```
def create
  #render plain: params[:article].inspect
  #This displays the params hash data in the browser
  @article = Article.new(article_params)
  #pulls the specified attributes from params with private method from bottom o
  @article.save
  redirect_to article_path(@article)
```

Of course, this only works after we create a **SHOW** action in the ArticlesController. Remember, the **SHOW** needs a template to display the new **Article** and the instructions on what exactly it is displaying:

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

With the redirect from **CREATE**, the new instance **params** will be pushed along to the show view.

A simple template will suffice for now.

```
<h2>Selected Article</h2>

<p><b>Title:</b> <i><%= @article.title %></i></p>
<p><b>Description:</b> <i><%= @article.description %></i></p>
```

renders as

Above is a very simplified explanation of params. Read more at:

https://github.com/rails/strong_parameters,
https://learn.co/lessons/rails-create-action-readme
http://edgeguides.rubyonrails.org/action_controller_overview.html

---

So far we have a very unforgiving method of creating new object instances. There is nothing to alert us if something goes wrong. Lets change that.

First lets take advantage of the validations we set in the model and use a conditional statement to return a flash alert. Change the **CREATE** action to:

```ruby
def create
  #render plain: params[:article].inspect
  #This displays the params hash data in the browser
  @article = Article.new(article_params)
  #pulls the specified attributes from params with private method from bottom of page
  if @article.save
    flash[:notice] = "Article was successfully created"
    redirect_to article_path(@article)
  else
    render :new
    #could also be 'new'
  end
end
```

If all the validations are met when the form is submitted, the instance is saved to the db, the app redirects to the show view, and the flash message will display *somewhere*.
else if the validations are not met, the form will render again...

But where does the flash message appear? You could add it to the show template- but if you add the ***flash target*** block to layouts/application.html.erb, you can use it anytime a view needs to display the flash message instead of adding a new target each time. Application.html.erb is the frame for your views. Its where your nav bar and footer goes, built around a tiny little line...

```
<%= yield %>
```

which acts as a **div** for the views to render in. By adding the ***flash target*** script (*the place your flash will pop up*) just above the **yield**, you can use it as the destination for flash messages from other views as well:

application.html.erb

Flash target

Yield

```
DOCTYPE html>
tml>
ead>
<title>Workspace</title>
<%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track' => true %>
<%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
<%= csrf_meta_tags %>
head>
ody>

<div id="wrap">
  <nav class="navbar navbar-default">
        <div class="container-fluid">
            <!-- Brand and toggle get grouped for better mobile display -->
            <div class="navbar-header">
                <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" da
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <ul><%= link_to 'ATLAS', root_path, class: "navbar-brand" %></ul>
            </div>
            <!-- Collect the nav links, forms, and other content for toggling -->
            <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
                <ul class="nav navbar-nav navbar-right">
                    <li><%= link_to 'HOME', root_path %></li>
                    <li><a href="#">XXXXXX</a></li>
                    <li><%= link_to 'ABOUT', about_path %></li>
                    <li><a href="#">CONTACT</a></li>
                </ul>
            </div>
            <!-- /.navbar-collapse -->
        </div>
        <!-- /.container-fluid -->
  </nav><!--Nav ends-->

  <div class="container content-container">

    <% flash.each do |name, msg| %>
      <ul>
        <li><%= msg %></li>
      </ul>
    <% end %>
    <!--this flash block will render flash messages from views and partials in the app wrapper-

    <%= yield %>

  </div>

</div><!--Wrap ends-->


<div id="footer">
  <div class="container footer-container">
    <p class="credits">&copyATLAS:  Another "Twitter-Like App" Site</p>
  </div>
</div>
                                                    45:97  HTML (Ruby)  Spaces: 2
```

Now you have a flash message when you are successful, but what about when your submitted data doesn't meet the validations?

We already gave a conditional that says to render the '**new**' page again when errors are present, so lets take advantage of the fact that the page is read top to bottom and inject another conditional script that looks for these errors and then displays them to the view:

new.html.erb

```
<h2>Create an article</h2>

<% if @article.errors.any? %>
<h5 style="color:red">The following error(s) prevented the article from being created</h5>
  <% @article.errors.full_messages.each do |msg| %>
   <li><%= msg %></li>
  <% end %>
<% end %>
<!--if errors occur based on validations, will return those errors as message-->


<%= form_for @article do |f| %>
  <p>
    <%= f.label :title %><br/> <!--line breaks for for layout-->
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :description %><br/> <!--line breaks for for layout-->
    <%= f.text_area :description %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

If there any errors related to the new attempt, @article, display each message on its own line...

After the messages display, the new form is rendered and try again.

Editing Instances from the Browser

As with **CREATE**, **EDIT** needs an action and template. The route was created in routes.rb with resources :articles. In the template, we can use the form and the errors block that we used in CREATE as well. Then just pass the object params through, like the SHOW action. Rails will populate the model backed form with params.

Submitting now earns a big fat error message:

PATCH needs directions from **UPDATE**:

```
def update
  @article = Article.find(params[:id])
  if @article.update
    flash[:notice] = "Article was successfully updated"
    redirect_to article_path(@article)
  else
    render :edit
  end
end
```

wrong number of arguments (given 0, expected 1)

Extracted source (around line #31):

```
29    def update
30      @article = Article.find(params[:id])
31      if @article.update
32        flash[:notice] = "Article was successfully updated"
33        redirect_to article_path(@article)
34      else
```

Now when we submit, there is an error we haven't seen before:

It seems to be saying that @article.update needs an argument???

Remember with CREATE we had the private method, *article_params*, passing the specific attributes we wanted sent to the db. UPDATE needs this info too. Use *article_params* as the argument.

Now the edit is complete.

Viewing a list of all instance articles the Browser

For a complete list rake routes shows *articles GET /articles(.:format) articles#index*. Now its action/ argument and template time...

```
def index
  @article = Article.all
end
```

index.html.erb

```
1  <h3>Articles</h3>
```

We can display the Article's in a table for now and call each of them with a block within the table:

```
<h3>Articles</h3>

<table>
  <tr>
    <th>Title</th>
    <th>Description</th>
  </tr>

  <% @articles.each do |article| %>
  <tr>
    <td width="300px"><%= article.title%></td>
    <td><%= article.description%></td>
  </tr>
  <% end %>
</table>
```

```
<h3>Articles</h3>

<p><%= link_to "Create new article", new_article_path %></p>

<table>
  <tr>
    <th>Title</th>
    <th>Description</th>
  </tr>

  <% @articles.each do |article| %>
  <tr>
    <td width="350px"><%= article.title%></td>
    <td width="600px"><%= article.description%></td>
    <td width="30px"><%= link_to "Edit", edit_article_path(article) %></td>
    <td><%= link_to "Show", article_path(article) %></td>
  </tr>
  <% end %>
</table>
```

What if we want to edit an instance? We can use a helper method and link_to edit, delete and create.

*The block passes the params- only the block variable needs to be passed as an argument

```
<%= link_to "Back to articles listing", articles_path %>
<%= link_to "Edit this article", edit_article_path(@article) %>
```

Then add links to show.html.erb, new.html.erb and edit.html.erb where needed

Destroy & Partials

Before we add a DESTROY action and utilize it with our forms, lets take a look at the redundancy between edit.html.erb and new.html.erb. Both templates look exactly alike. Since Rails is all about DRY(*don't repeat yourself*) code, we will extract and copy the code to a form **partial**. We then call the **partial** in each view where the code is rendered. In the future, we start with the **partial** so we aren't writing code over and over again...

Partials templates that are there to avoid redundancies.

_form.html.erb

```
<h2>Create an article</h2>

<% if @article.errors.any? %>
<h5 style="color:red">The following error(s) prevented the article from being created</h5>
  <% @article.errors.full_messages.each do |msg| %>
    <li><%= msg %></li>
  <% end %>
<% end %>
<!--if errors occur based on validations, will return those errors as message-->



<%= form_for @article do |f| %>
  <p>
    <%= f.label :title %><br/> <!--line breaks for for layout-->
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :description %><br/> <!--line breaks for for layout-->
    <%= f.text_area :description %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
<!--FOR NEW.HTML.ERB: articles/new.html.erb form.  This data stored in params and
<!--FOR EDIT.HTML.ERB: articles/edit.html.erb form.  Edit this data, from params

<%= link_to "Back to articles listing", articles_path %>
```

Create a new file views/articles/_form.html.erb. A partial always starts with an underscore. Copy everything but the header of both the 'edit' and 'new' views, and then paste it into the new partial. To render the **partial**, just call it as shown below:

edit.html.erb

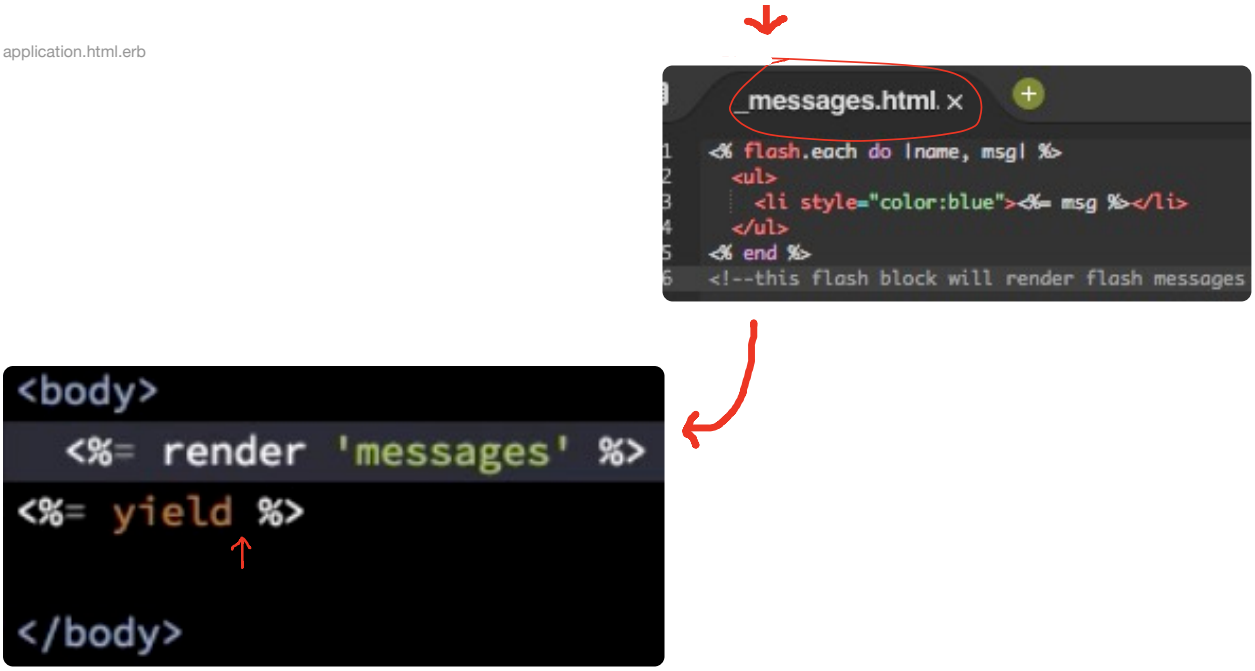
```
<h2>Edit an article</h2>

<%= render 'form' %>
```

Reload and navigate to the 'new' and 'edit' pages... Everything should work the same.

Remember the flash message sent from the CREATE and UPDATE actions in the ArticlesController to the views layout wrapper, application.html.erb? Lets put the messages *target block* in a **partial** called _messages.html.erb, and then call it above yield...

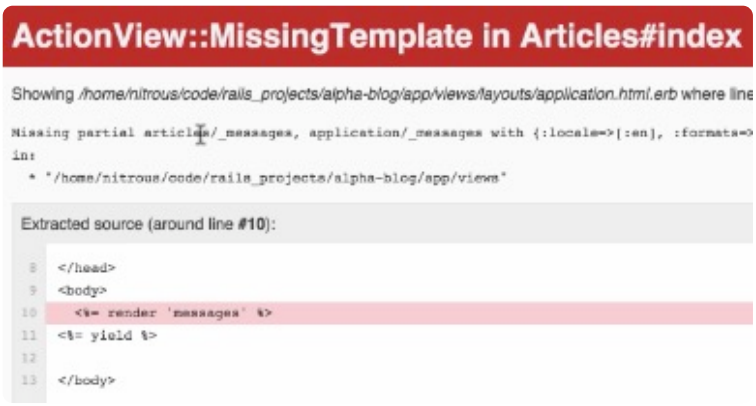
```
<body>
  <% flash.each do |name, msg| %>
    <ul>
      <li><%= msg %></li>
    </ul>
  <% end %>
<%= yield %>

</body>
```

application.html.erb



application.html.erb

Now when we submit a new Article instance, it runs through the conditional in the CREATE action. If the instance meets all the validations set in the **model**, the CREATE action fires off an alert that is picked up by layouts/_messages.html.erb. We call _messages.html.erb in layouts/application.html.erb with *render* and...



...we get an error message. WTF?

This happens because application.html.erb doesn't search its own folder for data, views/layouts/, it looks to the views folder corresponding to the controller. Here we have ArticlesController doing all the talking, which means application.html.erb is looking for data in views/articles/. All we need to do is tell application.html.erb where the partial is located:

```
<%= render 'layouts/messages' %>
```

Keep application.html.erb clean- use partials for the navbar and the footer too.

Good to go! Now lets add a delete functionality. Check rake routes:

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

We know a DESTROY action is needed that includes all the steps to remove data and redirect the user:

```
def destroy
  @article = Article.find(params[:id])
  @article.destroy
  flash[:notice] = "Article was successfully deleted`"
  redirect_to articles_path
end
```

```
<td><%= link_to "Show", article_path(article) %></td>
<td><%= link_to "Delete", article_path(article) %></td>
```

But the helper path is the same as SHOW, so how do we call the right method?

We actually have to specify the HTTP request DELETE... Its also a good idea to add a confirmation so that users don't accidentally lose data.

```
<td><%= link_to "Delete", article_path(article), method: :delete, data: {
 confirm: "Are you sure?" } %></td>
```

Go ahead and test the new link and delete something- Did Rails give you a pop-up confirmation? Good.

Now would be a good time to add the articles_path to the applications navbar.

Lets do a little bit of house keeping...

In articles_controller.rb, you may have noticed the line `@article = Article.find(params[:id])` shows up quite a bit. Between the various actions, it is used 4 times. This makes it a prime candidate for its own method... Under **private**, lets wrap this line in a method called **set_article**:

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

Now replace each of these lines, in each article where it is found, with **set_article**. If you check the app now, it should run if you make sure set_article is called first in each action. But we want to make sure that regardless of what is added to the controller, set_article will always be called first, but only in its own method. For this we add:

```
class ArticlesController < ApplicationController
  before_action :set_article, only: [:edit, :update, :show, :destroy]
```

Our production environment is set up and we have successfully pushed changes to Heroku (git push heroku master). Open the app via heroku open and you will notice that only the static aspects are working correctly. Our database table does not exist in production. To push the migrations to Heroku run heroku run rake db:migrate- watch as the table is built in the terminal. Go ahead and add some articles in the new production app.

Styling with Bootstrap

To style the app with bootstrap, we first need to download the bootstrap-sass gem. See the docs at : github.com/twbs/bootstrap-sass

Follow the instructions to install...

You can update the gem in Gemfile if necessary. Run bundle install and check the version with bundle show gem-name

At bootstrap.com/components you can find various ready built page properties- Navbars, jumbotrons, etc. to copy and paste. Just fill in the parts that are specific to your app.

See Bootstrap notes.

Change Rails default **new/edit form** to bootstrap...

```
<div class="row">
  <div class="col-xs-12">
    <%= form_for(@article, :html => {class: "form-horizontal", role: "form"}) do |f| %><!--form class="form-horizontal">-->
      <div class="form-group">
        <div class="control-label col-sm-2"><!--<label for="inputEmail3" class="col-sm-2 control-label">Email</label>-->
          <%= f.label :title %>
        </div>
        <div class="col-sm-8">
          <%= f.text_field :title, class: "form-control", placeholder: "Title of article", autofocus: true %><!--<input type
            ="email" class="form-control" id="inputEmail3" placeholder="Email">-->
        </div>
      </div>
      <div class="form-group">
        <div class="control-label col-sm-2">
          <%= f.label :description %>
        </div>
        <div class="col-sm-8">
          <%= f.text_area :description, rows: 10, class: "form-control", placeholder: "Description", autofocus: true %>
        </div>
      </div>
      <div class="form-group">
        <div class="col-sm-offset-2 col-sm-10">
          <%= f.submit class: "btn btn-primary btn-lg" %>
        </div>
      </div>
    <% end %>

    <%= link_to "Back to articles listing", articles_path %>
  </div>
</div>
<!--FOR NEW.HTML.ERB: articles/new.html.erb form.  This data stored in params and sent through CREATE action via POST-->
<!--FOR EDIT.HTML.ERB: articles/edit.html.erb form.  Edit this data, from params and submitted via PATCH-->
```

errors message block styled separately

+To mix Rails and bootstrap, the form_for block:

```
<%= form_for @article do |f| %>
```

is injected with bootstrap html "form-horizontal" class and "form" role. These are the classes you see in the default bootstrap form, and that keep the form aligned and spaced horizontally once rendered.

+The 'offset' class moves the element horizontally, much like margin in css.

+Sets number of vertical rows

+The navigation already has a button to take the user to the main articles page, so lets use this link as a cancel button:

```
<div class="col-xs-4 col-xs-offset-4">
  [<%= link_to "Cancel request and return to articles listing", articles_path %>]
</div>
```

Bootstrap styled alerts and messages:

Before Bootstrap:

```
<% if @article.errors.any? %>
<h5 style="color:red">The following error(s) prevented the article from being created</h5>
  <% @article.errors.full_messages.each do |msg| %>
    <li><%= msg %></li>
  <% end %>
<% end %>
<!--if errors occur based on validations, will return those errors as message-->
```

After Bootstrap:

```
<% if @article.errors.any? %>
  <div class="row">
    <div class="col-xs-8 col-xs-offset-2">
      <div class="panel panel-danger"><!--sets red alert color-->
        <div class="panel-heading">
          <h2 class="panel-title">
            <%= pluralize(@article.errors.count, "error") %><!--method to display 'error' or 'errors' based on count-->
            Prohibited this article from being saved: <!--message header-->
          </h2>
          <div class="panel-body"><!--block body/ main-->
            <ul>
              <% @article.errors.full_messages.each do |msg| %>
                <li><%= msg %></li>
              <% end %>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </div>
<% end %>
<!--if errors occur based on validations, will return those errors as message-->
```

This errors script can be used anywhere if we create a partial. See Rails notes for instruction.

After styling the "errors" alert block for **edit/new.html.erb** with bootstrap, lets set up a partial that can be used by any view to display errors in the future. For partials that are not view specific, we create a **views/share** folder. In the new share folder, create a partial **_errors.html.erb**, and add the errors block:

_form.html.erb

```
<%= render "share/errors" %>

<div class="row">
  <div class="col-xs-12">
    <%= form_for(@article, :html => {class: "form-horizontal", role
      : "form"}) do |f| %><!--form class="form-horizontal">-->
      <div class="form-group">
        <div class="control-label col-sm-2"><!--<label for
          ="inputEmail3" class="col-sm-2 control-label">Email
          </label>-->
          <%= f.label :title %>
        </div>
        <div class="col-sm-8">
          <%= f.text_field :title, class: "form-control",
            placeholder: "Title of article", autofocus: true %><!
            --<input type="email" class="form-control" id
            ="inputEmail3" placeholder="Email">-->
        </div>
      </div>
      <div class="form-group">
        <div class="control-label col-sm-2">
          <%= f.label :description %>
        </div>
```

_errors.html.erb

```
<% if @article.errors.any? %>
  <div class="row">
    <div class="col-xs-8 col-xs-offset-2">
      <div class="panel panel-danger"><!--sets red alert color-->
        <div class="panel-heading">
          <h2 class="panel-title">
            <%= pluralize(@article.errors.count, "error") %><!--method to display
              'error' or 'errors' based on count-->
            Prohibited this article from being saved: <!--message header-->
          </h2>
          <div class="panel-body"><!--block body/ main-->
            <ul>
              <% @article.errors.full_messages.each do |msg| %>
                <li><%= msg %></li>
              <% end %>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </div>
<% end %>
<!--if errors occur based on validations, will return those errors as message-->
```

Link the errors partial to _form.html.erb and voilá. Test the forms in the browser to get:

Everything should be working nicely, but this is a resource shared throughout the app... What if we want to use it with a different MVC in the future- lets say @user?? In that case, the instances of @article will create problems. Lets fix it...

Replace each instance call with the Rails method obj, like this:

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

=>

```
<% if obj.errors.any? %>
```

Now we call the preferred object where the partial is rendered:

```
<%= render "share/errors", obj: @article %>
```

We can do the same thing for the alerts in the layouts/_messages.html.erb...

Before Bootstrap:

```
<% flash.each do |name, msg| %>
  <ul>
    <li style="color:blue"><%= msg %></li>
  </ul>
<% end %>
<!--this flash block will render flash messages when called in a template-->
```

After Bootstrap:

```
<div class="row">
  <div class="col-xs-10 col-xs-offset-1">
    <% flash.each do |name, msg| %>
      <div class='alert alert-<%= "#{name}" %>'><!--the alert-type determines bootstrap alert color.  Here we
        pass :notice from the article controller-->
        <a href="#" class="close" data-dismiss="alert">&#215;</a>
        <%= content_tag :div, msg, :id => "flash_#{name}" if msg.is_a?(String) %>
      </div>
    <% end %>
  </div>
</div>
<!--this flash block will render flash messages when called in a template-->
```

The alert-type determines the color displayed with the alert. The name is passed through the block, from:

```
flash[:notice] = "Article was successfully created"
```

Along with **notice**(no color), there is **success**(green), **info**(blue), **warning**(yellow), and **danger**(red)

Here the alert includes an "X" to close it out, in the form of a link that doesn't go anywhere. **&#215** == **X**

The **content_log** line creates an id for the message div, as long as the message is a string

Bootstrap style show view

```
<h2>Selected Article</h2>

<p><b>Title:</b> <i><%= @article.title %></i></p>
<p><b>Description:</b> <i><%= @article.description %></i></p>

<%= link_to "Back to articles listing", articles_path %> |
<%= link_to "Edit this article", edit_article_path(@article) %>
```

Before Bootstrap

```
<h2 align="center">Title:   <i><%= @article.title %></i></h2>
<div class="well col-xs-8 col-xs-offset-2">
  <h4 class="center description" align="center"><strong>Description:</strong></h4>
  <hr>
  <%= simple_format(@article.description) %>
  <div class="article-actions">
    <%= link_to "Edit this article", edit_article_path(@article), class: "btn btn-xs btn-primary" %>
    <%= link_to "View all articles", articles_path, class: "btn btn-xs btn-primary" %>
    <%= link_to "Delete this article", article_path(@article), method: :delete,
                data: { confirm: "Are you sure?" },
                class: "btn btn-xs btn-danger" %>
  </div>
</div>
```

After Bootstrap

Bootstrap style index view

```
<h3 align="center">Articles</h3>

<p><%= link_to "Create new article", new_article_path %></p>

<table>
  <tr>
    <th>Title</th>
    <th>Description</th>
  </tr>

  <% @articles.each do |article| %>
  <tr>
    <td><%= link_to "Edit", edit_article_path(article) %></td>
    <td><%= article.description%></td>
    <td><%= link_to "Show", article_path(article) %></td>
    <td><%= link_to "Delete", article_path(article), method: :delete, data: { confirm: "Are you
      sure?" } %></td>
    <td width="350px"><%= article.title%></td>
  </tr>
  <% end %>
</table>
```

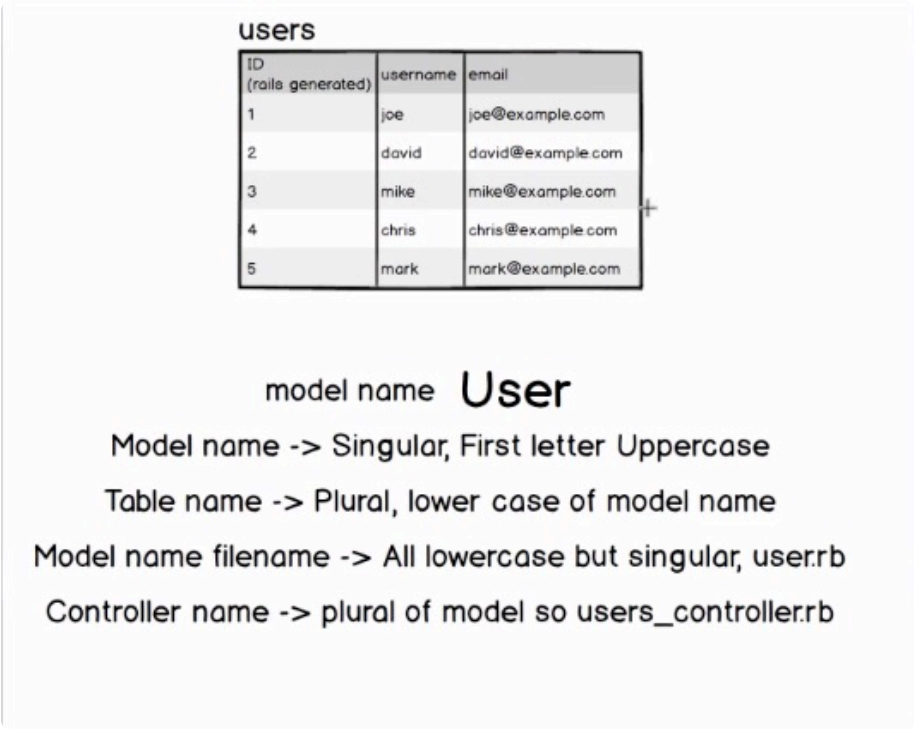
Before Bootstrap

```
<h3 align="center">Articles</h3>


<% @articles.each do |article| %>
  <div class="row">
    <div class="col-xs-8 col-xs-offset-2">
      <div class="well well-lg">
        <div class="article-title">
          <p><b>Title: </b><%= link_to article.title, article_path(article) %></p>
        </div>
        <div class="article-body">
          <p><b>Description: </b><%= truncate(article.description, length: 100) %></p> <!
            --truncate displays the amount of characters specified by length-->
        </div>
        <div class="article-actions">
          <%= link_to "Edit", edit_article_path(article), class: "btn btn-xs btn-primary" %>
          <%= link_to "Delete", article_path(article), method: :delete,
                      data: { confirm: "Are you sure?" },
                      class: "btn btn-xs btn-danger" %>
        </div>
      </div>
    </div>
  </div>
<% end %>
```

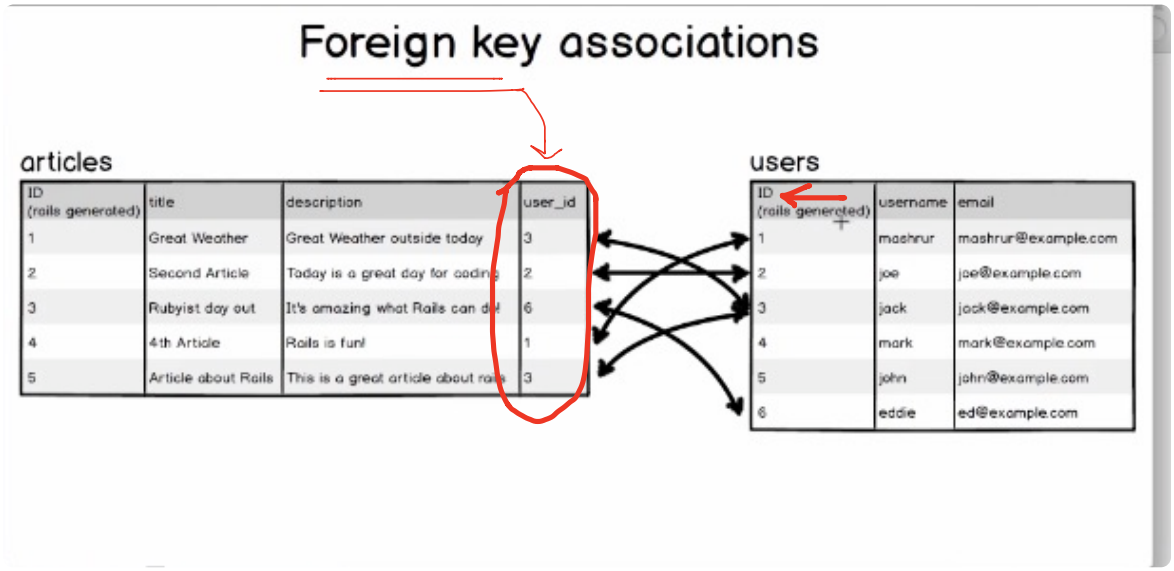
After Bootstrap **see truncate in clip**

Make sure changes are committed to the remote repo and push changes to Heroku. Run rake as well.

Time To add Users MVC



Our ***users*** will have a **username** and an **email**. The ***User*** will be able to read and write articles, so how do we keep track of which article belongs to which user? This is what **associations** are for:



**Also read http://guides.rubyonrails.org/association_basics.html

Associations are all about linking two db models by creating a relationship between an instance **ID** and a **foreign key**. This concept is independent of rails and seen in most relational data.

This is a **belongs_to : has_many** relationship.

Lets first demo this in the test-app:

run rails generate scaffold User username:string email:string

then rake db:migrate to create the table

Check the new migration file and schema data for a visual of what you just did- Or enter User into Rails Console to see attributes. Go ahead and create some user instances, **user = User.new(username: "name", email: "email@example.com")**

Since scaffolding was used, the UsersController already contains the actions data just like we created in ArticlesController.

Lets create a comments MVC so we can associate the users table with the comments table:

run rails generate scaffold Comment description:text user:references

Since we already know we want to associate the two models, we can use the name of the table we want to link with a **foreign key,** and the keyword '***references***' (highlighted) in the scaffolding command. Now the comments table will have a **user_id** attribute- which means each comment will belong to that user. Check the new migration file:

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.text :description
      t.references :user, index: true, foreign_key: true

      t.timestamps null: false
    end
  end
end
```

Don't forget to run the migration.

```
=> Comment(id: integer, description: text, user_id: integer, created_at: datetime, updated
_at: datetime)
```

You can undo a generate scaffold with rails destroy scaffold Comment

---

The Comment model is generated with the *belongs_to* method:

```
class Comment < ActiveRecord::Base
  belongs_to :user
end
```

```
class User < ActiveRecord::Base
    has_many :comments
end
```

But remember, the association goes both ways... We need to manually update the User model. If a comment belongs_to a user, a user has_many comments:

This allows us to query *comments* through *users*:

```
2.3.0 :005 > user = User.first
  User Load (0.4ms)  SELECT  "users".* FROM "users"  ORDER BY "users"
."id" ASC LIMIT 1
 => #<User id: 1, username: "Bob", email: "bob@mail.com", created_at:
 "2016-09-15 22:59:25", updated_at: "2016-09-15 22:59:25">
2.3.0 :006 > user.comments
  Comment Load (0.4ms)  SELECT "comments".* FROM "comments" WHERE "co
mments"."user_id" = ?  [["user_id", 1]]
 => #<ActiveRecord::Associations::CollectionProxy []>
```

**In this particular case there are no comments

To create a new comment:

```
2.3.0 :007 > comment = Comment.new(description: "This is a new comment from
user one.", user: user)
 => #<Comment id: nil, description: "This is a new comment from user one.",
user_id: 1, created_at: nil, updated_at: nil>
```

Here we specify which user this comment will belong to- In this case, we set the variable 'user' to the *first user.*

Ta-da!!!

```
2.3.0 :015 > user.comments
  Comment Load (0.3ms)  SELECT "comments".* FROM "comments" WHERE "comments"
."user_id" = ?  [["user_id", 1]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Comment id: 1, descript
ion: "This is a new comment from user one.", user_id: 1, created_at: "2016-0
9-16 00:56:04", updated_at: "2016-09-16 00:56:04">]>
```

You can also call *comment* to see the new comment or *comment.user* to see who created the new comment.

From here on, the master branch should always be deployable. Any changes to the app should be written on a branch- See Git Cheats(notes) or https://www.atlassian.com/git/tutorials/using-branches/git-branch

The Comment and User MVC's will be created on a branch called create-users

Lets manually generate the MVC from above for the Alpha-blog.

First User:

rails generate migration create_users

rake db:migrate

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :username
      t.string :email
      t.timestamps
    end
  end
end
```

models/user.rb

```
class User < ActiveRecord::Base

end
```

Check the rails console, like above. If all the CRUD actions are working, add the branch to master:

git add and commit
git checkout master (back to master)
git merge branch-name (pull changes in)
git branch -d create-users (optional, only works if changes merged)

git push

Now to add validations to the model. Don't forget to branch.

Validations for User class:

- username must be present and unique

- email must be present and unique

- validate email format using regex

For a list of validations see http://guides.rubyonrails.org/active_record_validations.html

Some new validation methods were used for this class. More on regex at rubular.com

```
class User < ActiveRecord::Base
  validates :username, presence: true,
                       uniqueness: { case_sensitive: false },
                       #ensures unique username + no case sense (joe = Joe = jOE)
                       length: { minimum: 3, maximum: 25 }
  VALID_EMAIL_REGEX= /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 105 },
                       uniqueness: { case_sensitive: false },
                       format: { with: VALID_EMAIL_REGEX}
                       #using regex for proper email address format
end
```

Check the validations in rails console- use *variable.valid?* and *variable.errors.full_messages* to see results.

The User class is set up, lets add the associations with the Article class. The first thing needed is a **foreign key** attribute for the articles table.

- create a rails migration
- add the new attribute to the migration file
- run rake

```
class AddUserIdToArticles < ActiveRecord::Migration
  def change
    add_column :articles, :user_id, :integer
  end
end
```

Check schema.rb to see if the table was updated.

Now set association:

```
class Article < ActiveRecord::Base
  belongs_to :user
  validates :title, presence: true, length: { minimum: 3, maximum: 30 }
  validates :description, presence: true, length: { minimum: 10, maximum: 100}
  validates :user_id, presence: true
end
```

added validation

```ruby
class User < ActiveRecord::Base
  has_many :articles
  before_save { self.email = email.downcase }
  #saves email in downcase
  validates :username, presence: true,
            uniqueness: { case_sensitive: false },
            #ensures unique username + no case sense (joe = Joe = jOE)
            length: { minimum: 3, maximum: 25 }
  VALID_EMAIL_REGEX= /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 105 },
            uniqueness: { case_sensitive: false },
            format: { with: VALID_EMAIL_REGEX}
            #using regex for proper email address format
end
```

Now when you create an article instance in rails console, it will require a **user_id**:

```
2.3.0 :003 > article = Article.new(title: "Another article", description: '
Lorem ipsum another article', user: User.first)
```

Remember, Rails is super smart. It will allow you to use **user:** instead of **user_id:**, when referencing the articles attribute. When **user_id:** is used, the input needs to be an actual user **id** number (see ***User.all***).
To see a list of only **username's and **id**'s use *User.pluck :username, :id*

If we try to create an article through the dev app an error message will appear. Our validations are looking for the **user_id**, but our form is only sending **id**(default), **title**, **description**, and **time/date**.

Lets try and create an article, but lets also use this attempt to touch on how to use the **byebug** gem:

The byebug gem is in the Gemfile under group :development, :test do

Since we are using the create action in our failed attempt to create an article, add the debugger method to the top of the action:

```ruby
def create
  debugger
  #render plain: param
```

Now attempt to create an article and watch the terminal where the server is running... byebug has paused the server and now you can see everything being passed in. Try typing *article_params* into the CLI:

```
Processing by ArticlesController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"kkeffFCl9Sl68c8c/Bb7IP/4cNeXcj
5UIhYEdDYRCmlZtpilWSsMg5WCv6bX7rXRWivllVp8MlAIatT7/MAhsA==", "article"=>{"title"=
>"bbbb bbb bb", "description"=>"bbbbbbbb ddddddd sssss"}, "commit"=>"Create Artic
le"}

[19, 28] in /home/ubuntu/workspace/app/controllers/articles_controller.rb
   19:
   20:   def create
   21:     debugger
   22:     #render plain: params[:article].inspect
   23:     #This displays the params hash data in the browser
=> 24:     @article = Article.new(article_params)
   25:     #@article.user = User.first
   26:     #pulls the specified attributes from params with private method from b
ottom of page
   27:     if @article.save
   28:       flash[:info] = "Article was successfully created"
(byebug)
```

```
(byebug) article_params
{"title"=>"bbbb bbb bb", "description"=>"bbbbbbbb ddddddd sssss"}
(byebug)
```

This is a convenient way to see the data being passed- and notice that there is no **user_id**, so we know this will not save.

Ctrl + d returns to server and unpauses

Another convenient debugger we can add in the views wrapper, layouts/application.html.erb, is:

```
</div><!--Wrap ends-->

<%= render 'layouts/footer' %>
<%= debug(params) if Rails.env.development? %>
```

This provides some basic info in the browser, which can help when running into errors:

```
--- !ruby/hash-with-ivars:ActionController::Parameters
elements:
  controller: articles
  action: show
  id: '6'
ivars:
  :@permitted: false
```

In the meantime, we can use **@article.user** = **User.first** as a temporary way to include a **user_id** with a new article.

Lets add user data to each article, starting with articles/index.html.erb.

```
<small>Created by: <% if article.user %>
                      <%= article.user.username %>
                    <% else %>
                      <i>unknown</i>
                    <% end %>
                    <%= time_ago_in_words(article.created_at) %> ago, last updated:
                    <%= time_ago_in_words(article.updated_at) %> ago </small>
```

Because many of our Article instances were created before the User class, calling article.user.username on these earlier articles will return an error since there is no **user_id**.

To fix this we use a conditional that prints "unknown" in the absence of this user data.

After the conditional we include time stamps. The time_ago_in_words method provides an easily readable timestamp.

Now that the **foreign key** attribute is present, every new article gets a **user_id**. If you were to delete all the articles lacking the **user_id**, this conditional would be pointless.

Adding Secure Passwords

Users require an individual identifier. We already set user validations to only allow one of each username or email, so either will work as the identifier. We will be using email along with the Rails *has_secure_password* method, http://api.rubyonrails.org/classes/ActiveModel/SecurePassword/ClassMethods.html. It requires:

- **bcrypt** gem
- **password_digest** attribute in the users table.
- *has_secure_password* method in the model

```
class User < ActiveRecord::Base
  has_many :articles
  before_save { self.email = email.downcase }
  #saves email in downcase
  validates :username, presence: true,
            uniqueness: { case_sensitive: false },
            #ensures unique username + no case sense (
            length: { minimum: 3, maximum: 25 }
  VALID_EMAIL_REGEX= /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z
  validates :email, presence: true, length: { maximum:
            uniqueness: { case_sensitive: false },
            format: { with: VALID_EMAIL_REGEX}
            #using regex for proper email address form
  has_secure_password
end
```

```
create_table "users", force: :cascade do |t|
  t.string   "username"
  t.string   "email"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.string   "password_digest"
end
```

schema.rb

Here is how it works:

```
2.3.0 :039 > user = User.find(2)
  User Load (0.5ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT 1  [["i
d", 2]]
 => #<User id: 2, username: "Joe", email: "joe@mail.com", created_at: "2016-09-17 23:49:
58", updated_at: "2016-09-18 00:05:35", password_digest: nil>
2.3.0 :040 > user.password = 'password'
 => "password"
2.3.0 :041 > user.save
   (0.1ms)  begin transaction
  User Exists (0.3ms)  SELECT  1 AS one FROM "users" WHERE (LOWER("users"."username") =
LOWER('Joe') AND "users"."id" != 2) LIMIT 1
  User Exists (0.1ms)  SELECT  1 AS one FROM "users" WHERE (LOWER("users"."email") = LOW
ER('joe@mail.com') AND "users"."id" != 2) LIMIT 1
  SQL (0.5ms)  UPDATE "users" SET "password_digest" = ?, "updated_at" = ? WHERE "users".
"id" = ?  [["password_digest", "$2a$10$yG6uiPvmdXsnVW/dZe0sXut/bZ6ihZGvWZwDlyiKHaEyJWXFA
bIZC"], ["updated_at", "2016-09-20 03:45:15.862915"], ["id", 2]]
   (40.6ms)  commit transaction
 => true
2.3.0 :042 >
```

First the the password is saved to the user. The **password** method is used here, "digest" is dropped.

Notice that the **password_digest** attribute is a long mess of digits. This is the "*one way hash digest*" that converts the password to a unique string of digits for each entry into the db.

Try some authentications:

```
2.3.0 :042 > user.authenticate 'ppassword'
 => false
2.3.0 :043 > user.authenticate 'password'
 => #<User id: 2, username: "Joe", email: "joe@mail.com", created_at: "2016-09-17 23:49:
58", updated_at: "2016-09-20 03:45:15", password_digest: "$2a$10$yG6uiPvmdXsnVW/dZe0sXut
/bZ6ihZGvWZwDlyiKHaE...">
```

If the db is ever infiltrated, the hacker can't get to the actual passwords.

SignUp Functionality

```
get 'signup', to: 'users#new'
```

The signup page should be at https://.....com/signup, so lets create that route at:

And then set up the UserController. If the user is signing up, then a new action and a model-backed form are needed.

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end
end
```

We set up a form for the Article class. just update it to fit this class for new.html.erb with **username**, **email** and **password** attributes.

```
<%= render "share/errors", {obj: @user} %>


<div class="row">
  <div class="col-xs-12">
    <%= form_for(@user, :html => {class: "form-ho
      <div class="form-group">
        <div class="control-label col-sm-2"><!--
          <%= f.label :username %>
        </div>
        <div class="col-sm-8">
          <%= f.text_field :username, class: "for
        </div>
      </div>
      <div class="form-group">
        <div class="control-label col-sm-2">
          <%= f.label :email %>
        </div>
        <div class="col-sm-8">
          <%= f.email_field :email, class: "form-
        </div>
      </div>
      <div class="form-group">
        <div class="control-label col-sm-2">
          <%= f.label :password %>
          <!--'password' is the virtual attribute
        </div>
        <div class="col-sm-8">
          <%= f.password_field :password, class:
```

To create a new User object, we still need a route and create action for the **POST** method:

Route(either):

post 'users', to: 'user#create'

-or-

resources :users, except: [:new]

```
class UsersController < ApplicationController

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      flash[:info] = "Welcome to Atlas, #{@user.username}!"
      redirect_to articles_path
    else
      render :new
    end
  end

  private

  def user_params
    params.require(:user).permit(:username, :email, :password)
  end

end
```

Now the MVC is complete and we can add users.

Don't forget the create action needs the User attributes passed in from *params.require* via a private action. See the ArticleController for a reminder on how to do this:

Steve Buscemi thinks you are doing a great job!

Try out byebug with debugger under the create action, then type in *params* command to see what is being passed to the db:

```
[2, 11] in /home/ubuntu/workspace/app/controllers/users_controller.rb
    2:
    3:   def new
    4:     @user = User.new
    5:   end
    6:
    7:   def create
    8:     debugger
=>  9:   end
   10:
   11: end
(byebug) params
{"utf8"=>"✓", "authenticity_token"=>"K2XMz0AbIMrlnT6JN1Kp25hgTTU25LEDZsTlpzmcnAzglMsW
SZXZYAruTjMcqucqPbPYd/vqvQdMuDUo80231Q==", "user"=>{"username"=>"brandon", "email"=>"
bb@example.com", "password"=>"password"}, "commit"=>"Sign up", "controller"=>"users",
 "action"=>"create"}
(byebug) Completed 500 Internal Server Error in 581913ms (ActiveRecord: 0.0ms)
```

Ctrl + d unpauses the server to complete the action.

Edit users

We used ***resources***: to set up the user routes, so **users#edit** is already available. We do need an edit action and view, and an update action:

```
def edit
  @user = User.find(params[:id])
end
```

This form will be almost identical to the form in new.html.erb, except for some differences in meta data.

Once we finish the update action, try editing a user in the browser:

```
def update
  @user = User.find(params[:id])
  if @user.update(user_params)
    flash[:info] = "Account successfully updated"
    redirect_to articles_path
  else
    render :edit
  end
```

Since both users/new.html.erb and users/edit.html.erb are using almost the same form, go ahead and put them in a partial...

There is a small difference between the forms- The edit submit button says "*Update*" and the new button says "*Sign up*":

```
<%= f.submit 'Update', class: "btn btn-primary btn-lg" %>
```

```
<%= f.submit 'Sign up', class: "btn btn-primary btn-lg" %>
```

Luckily rails has a useful method to handle this, new_record?, that we can use in a conditional:

```
<%= f.submit( @user.new_record? ? 'Sign up' : 'Update', cla
```

Check the browser pages and make sure everything is working.

User profile(show) page

- Show action
- View
- Style
  - For a profile image we will use a service called Gravatar(global avatar) that attaches an image to a user email. https://en.gravatar.com/

    Gravatar is a third party and we can create a method to insert an image. Anytime we create a method like this, we use app/helpers/application_helpers.rb.

    First we name & call our method inline:

```
<h1 align="center">Welcome to <%= @user.username %>'s page</h1>

<div class="row">
  <div class="col-md-4 col-md-offset-4" align="center">
    <%= gravatar_for @user %>
  </div>
</div>
<h4 align="center"><%= @user.username %>'s articles</h4>
```

    Then we create the method in application_helpers.rb:

```
module ApplicationHelper
  def gravatar_for(user)
    gravatar_id = Digest::MD5::hexdigest(user.email.downcase)
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}"
    image_tag(gravatar_url, alt: user.username, class: "img-circle")
    #this method can be found in Gravatar docs and is specific to Gravatar
  end
end
```

In this case the method is specific to Gravatar and can be found in the docs.

Save and reload the browser. Either an image or a Gravatar logo will appear in the users profile, depending on whether the user has a Gravatar account.

For sizing the image, add an ***options*** parameter to the *gravatar_id* method:

```
module ApplicationHelper
  def gravatar_for(user, options = {size: 80}) #parameter 2 is the default size.
    gravatar_id = Digest::MD5::hexdigest(user.email.downcase)
    size = options[:size]
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}?s=#{size}"
    image_tag(gravatar_url, alt: user.username, class: "img-circle")
    #this method can be found in Gravatar docs and is specific to Gravatar
  end
end
```

Then you can override it in the show view:

```
<h1 align="center">Welcome to <%= @user.username %>'s page</h1>

<div class="row">
  <div class="col-md-4 col-md-offset-4" align="center">
    <%= gravatar_for @user, size: 170 %>
    <!--size: 150 overrides the default size called in the grava
  </div>
</div>
<h4 align="center"><%= @user.username %>'s articles</h4>
```

Now lets display the users articles. We already have a page that displays *all* the articles, articles/index.html.erb, that can be shifted to a partial and then called to display only the articles the user created. Add articles/_article.html.erb, and call the partial on the index page.

This new partial works to call all the Article instances with @articles, since @articles is set to equal all articles in the index action. What if we want to change what this partial returns when it is called elsewhere?

1. Change @articles to a placeholder method:

```
<% obj.each do |article| %>
```

2. Define the placeholder when you call the partial:

```
<%= render 'article', obj: @articles %>
```

Now call the partial in the users show page with the correct definition to display only that users article instances.

```
<%= render 'articles/article', obj: @user.articles %>
```

**Remember, no need to call params. This calls the specific user and so only their articles

Create User index page + styles

```erb
<h1 align='center'>Users</h1>

<div align="center">
  <% @users.each do |user| %>
    <ul class="listing">
      <div class="row">
        <div class="well col-md-4 col-md-offset-4">
          <li><%= link_to gravatar_for(user), user_path(user) %></li>
          <!--gravatar image as link to users show page-->
          <li class="article-title user-title">
            <%= link_to user.username, user_path(user) %>
          </li>
          <li><small><%= pluralize(user.articles.count, "article") if user.articles %></small></li>
          <!--pluralize displays the number and pluralizes the keyword when necessary-->
        </div>
      </div>
    </ul>
  <% end %>
</div>
```

users/index.html.erb

```ruby
def index
  @users = User.all
end
```

Now we have a user index page that links to the individual users and displays their article instances. However, when we navigate to an articles a page, there is no user info. Lets fix that.

Simply take the **<ul>** portion of the index page:

```erb
<ul class="listing">
  <div class="row">
    <div class="well col-md-4 col-md-offset-4">
      <li><%= link_to gravatar_for(user), user_path(user) %></li>
      <!--gravatar image as link to users show page-->
      <li class="article-title user-title">
        <%= link_to user.username, user_path(user) %>
      </li>
      <li><small><%= pluralize(user.articles.count, "article") if user.articles %></small></li>
      <!--pluralize displays the number and pluralizes the keyword when necessary-->
    </div>
  </div>
</ul>
```

Leave the block portion behind. Since we will transfer this to the articles show page, the @user block is no longer necessary. Instead, we have to call each instance of User through articles:

```
<ul class="listing">
  <div class="row">
    <div class="well col-md-4 col-md-offset-4">
      <li><%= link_to gravatar_for(@article.user), user_path(@article.user) %></li>
      <!--gravatar image as link to users show page-->
      <li class="article-title user-title">
        <%= link_to @article.user.username, user_path(@article.user) %>
      </li>
      <li><small><%= pluralize(@article.user.articles.count, "article") if @article.user.articles %
        ></small></li>
      <!--pluralize displays the number and pluralizes the keyword when necessary-->
    </div>
  </div>
</ul>
```

At this time the app should be fully functional between the User MVC and the Article MVC, with the exception of admin settings to control who can edit each article.

To set limits on how many articles are loaded to a page, we use **pagination...**

This is for more than just the user experience. When we view the articles index page, we are using @articles.all to pull each instance from the db. This is fine when there are only a handful of instances, but if there were thousands of instances this could really slow things down. We only want to pull down the instances we want for each page. This makes for easy viewing and less strain on the db.

How to set up the pagination:

1. This requires two new gems. One for Rails and one for Bootstrap:

   Run bundle install --without pagination

```
gem 'will_paginate', '3.0.7'
# Rails pagination
gem 'bootstrap-will_paginate', '0.0.10'
# Bootstrap pagination
```

```
def index
  @articles = Article.paginate(page: params[:page], per_page: 10)
end
```

2. Add the **paginate params** in place of calling ***all*** instances. The default is **params[:page]**, which calls 20 instances. By including the *per_page* method we can customize.

This populates the view, but doesn't provide a way to get to the next page...

3. Use the *will_paginate* method in the index view and provide page to page navigation:

```
<div align="center">
  <%= will_paginate %>
</div>
<%= render 'article', obj: @articles %>
<div align="center">
  <%= will_paginate %>
</div>
```

Looking good! To paginate the users index page works the exact same way. If we want to paginate the articles that show under the users show page, there are a few extra steps:

1. In users/show.html.erb create a variable that represents the user and all that users articles, as well as, the pagination:

```
@user_articles = @user.articles.paginate(page: params[:page], per_page: 5)
```

2. Now call the new variable to render the pagination:

```
<div align="center">
  <%= will_paginate @user_articles %>
</div>

<%= render 'articles/article', obj: @user_articles %>

<div align="center">
  <%= will_paginate @user_articles %>
</div>
```

We call the variable after the *will_paginate* method, because otherwise this method defaults to calling the current URL appending the page parameter (look in address bar, in this case @users)

Login Form

Once a login is configured, we can start to restrict a users actions based on whether they are logged in or not. For example, limiting the users ability to create or edit articles unless they have gone through the verification of logging in.

The login will piggy back on the data we saved to the db with sign-up, and use **username** and **password**.

Lets use .com/login as the URL and create a new controller for user sessions. Because logging in is not db dependent, go ahead and build this without scaffolding:
Sessions: http://www.theodinproject.com/ruby-on-rails/sessions-cookies-and-authentication

```
get 'login', to: 'sessions#new'
```

```
class SessionsController < ApplicationController
  def new
  end

  def create
  end

  def destroy
  end
end
```

For each session we need a **form view**/new, a way to **submit that form**/create, and a way to **end the session**/destroy.

Remember, we are not using scaffolding and we only have a route for the new action, as of yet...

Lets create the rest of the routes we need:

```
get 'login', to: 'sessions#new'
post 'login', to: 'sessions#create'
get 'logout', to: 'sessions#destroy'
```

Lets build a login form at **sessions/new.html.erb**

This form will be just like the signup form, so copy & paste the signup(users/new.html.erb) *form_for* block minus any surrounding divs:

sessions_control × new.html.erb ×

```
<h1 align="center">Login</h1>

<%= form_for(@user, :html => {class: "form-horizontal", role: "form"}) do |f| %><!--form class="form-ho
  <div class="form-group">
    <div class="control-label col-sm-2"><!--<label for="inputEmail3" class="col-sm-2 control-label">Ema
      <%= f.label :username %>
    </div>
    <div class="col-sm-8">
      <%= f.text_field :username, class: "form-control", placeholder: "Enter user name", autofocus: tru
    </div>
  </div>
  <div class="form-group">
    <div class="control-label col-sm-2">
      <%= f.label :email %>
    </div>
    <div class="col-sm-8">
      <%= f.email_field :email, class: "form-control", placeholder: "Enter email" %>
    </div>
  </div>
  <div class="form-group">
    <div class="control-label col-sm-2">
      <%= f.label :password %>
      <!--'password' is the virtual attribute of 'password_digest'-->
    </div>
    <div class="col-sm-8">
      <%= f.password_field :password, class: "form-control", placeholder: "Enter password" %>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <%= f.submit( @user.new_record? ? 'Sign up' : 'Update', class: "btn btn-primary btn-lg") %>
    </div>
  </div>
<% end %>
```

Above the form still contains @user variables, but when we login we aren't creating anything new or accessing the db- The SessionsController is not "**model-backed**" and instead holds data in a **sessions hash** that is backed by the browser.

> "Sessions" are the idea that your user's state is somehow preserved when he/she clicks from one page to the next. Remember, HTTP is stateless, so it's up to either the browser or your application to "remember" what needs to be remembered.

Cont. http://www.theodinproject.com/ruby-on-rails/sessions-cookies-and-authentication

Above, @user references the new action that says it is equal to *User.new* and provides routing info for the block. We don't have this data in SessionsController and have to explicitly define the POST route:

from this:

```
<%= form_for(@user, :html => {class: "form-horizontal", role: "form"})
  do |f| %><!--form class="form-horizontal">-->
```

to this:

```
<%= form_for(:session, :html => {class: "form-horizontal", role:
  "form"}, url: login_path) do |f| %><!--form class="form-horizontal"
  -->
```

From here down tailor the form to fit a login screen:

```
<%= form_for(:session, :html => {class: "form-horizontal", role: "form"}, url: login_path) do |f| %><!--form
  class="form-horizontal"-->
  <div class="form-group">
    <div class="control-label col-sm-2">
      <%= f.label :email %>
    </div>
    <div class="col-sm-8">
      <%= f.text_field :email, class: "form-control", placeholder: "Enter user email", autofocus: true %>
    </div>
  </div>
  <div class="form-group">
    <div class="control-label col-sm-2">
      <%= f.label :password %>
      <!--'password' is the virtual attribute of 'password_digest'-->
    </div>
    <div class="col-sm-8">
      <%= f.password_field :password, class: "form-control", autocomplete: "off" %>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <%= f.submit "Login", class: "btn btn-primary btn-lg" %>
    </div>
  </div>
<% end %>
```

To see how Rails stores this data after we submit, temporarily add **render :new** to the create action and try logging a user in... It renders the page again, but take a look at the debugger we set up in the application.html.erb wrapper:

It displays the session hash that contains all the user data, for the user we logged in.

```
--- !ruby/hash-with-ivars:ActionController::Parameters
elements:
  utf8: "✓"
  authenticity_token: Bdqo0zPPggSvnsgPTG6bTd+JoblSwN72CtopQz5c
  session: !ruby/hash-with-ivars:ActionController::Parameters
    elements:
      email: brandon.brigance@gmail.com
      password: password
    ivars:
      :@permitted: false
  commit: Login
  controller: sessions
  action: create
ivars:
  :@permitted: false
```

You could also add *debugger* to the create action, and the byebug gem will pause the server and show the same info. Go ahead and remove the temporary line.

```
Processing by SessionsController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"t8q6K+NsIs2b4SJCLSGL5TEmpc5eZJZ6u9xdMZDmXK
B8O73y6uLbZ3SSUvgG2cUUlPUwjJNqmn6RoI2+Wjd3eQ==", "session"=>{"email"=>"brandon.brigance@gmail
.com", "password"=>"[FILTERED]"}, "commit"=>"Login"}
Return value is: nil
```

**params[:session] or params[:session][:email] gives more specific session data ctrl + d to unpause

Now we know what is being passed through create, we just have to tell create what to do with the data:

1. Use the session params to define **user**

```ruby
def create
  user = User.find_by(email: params[:session][:email].downcase)
end
```

2. Check if **user** was set and authenticate with the token.

```ruby
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
  # this validates whether defining 'user' was successful && authenticates it
```

3. If the auth. fails, reload the view and try again. In previous controllers the validations would provide a message that explains the failure. Because SessionsController is not model-backed, there is no validation to pull from and we must add a message.

```ruby
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
  # this validates whether defining 'user' was successful && authenticate

  else
    flash.now[:danger] = "There was a problem with your login attempt"
    render :new
end
```

Also notice flash *now*[....], rather than flash[....], is used to display the message. When just *flash* is used, the message will persist to the next view and should always be used when redirecting to another action via *redirect_to*. The *now* method is used when the

display is desired in the same view and rendering via the *render* method.

4. Add a success message and path.

```
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
  # this validates whether defining 'user' was successful && authentica
    flash[:info] = "You have successfully logged in"
    redirect_to users_path(user)
  else
    flash.now[:danger] = "There was a problem with your login attempt"
    render :new
end
```

5. This part is important. Everything else is set, now just save the users data from params to a **session**. Sessions are like cookies (see link above)

```
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    session[:user_id] = user.id
  # this validates whether defining 'user' was successful && authenticate
    flash[:success] = "You have successfully logged in"
    redirect_to user_path(user)
  else
    flash.now[:danger] = "There was a problem with your login attempt"
    render :new
  end
end
```

Try logging in now to test the messages and routes.

The **session** will pass along the user data until we logout/ run through the destroy action. Lets setup the destroy action now:

```
<li><%= link_to 'Logout', logout_path, method: :delete %></li>
```

6. Create a logout link in the navbar (or wherever), that follows the logout route. This runs through the destroy action and so *delete* must be called.

To destroy the **session**, just set the **session** to "***nil***" and configure the message and path:

```
def destroy
  session[:user_id] = nil
  flash[:info] = "You have logged out"
  redirect_to root_path
end
```

Now lets add authenticating methods to control restrictions throughout the app.

Because these authentications need to work throughout the app, we need to build them in the ApplicationController. Actions in ApplicationController are accessible by all the other controllers, but not to views. These actions will check that a user is see action before allowing certain processes within the app:

```ruby
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

  helper_method :current_user, :logged_in?

  def current_user

  end

  def logged_in?

  end

  def require_user

  end
end
```

But wait, we need access to these actions from the view? To do this, just turn each action into a helper method:

```ruby
def current_user
  User.find(session[:user_id]) if session[:user_id]
end
```

If the **session** contains a user id (so, if a user is logged in), then find that user in the User db.
We don't want to search the db every time we come across the user- that would just slow things down...

```ruby
def current_user
  @current_user ||= User.find(session[:user_id]) if session[:user_id]
end
```

Instead, if we already searched the db and nothing has changed, just return the same user.

** "@current_user **||=** *some_expression*" is the same as "@current_user **=** @current_user **||** *some_expression*"

```ruby
def logged_in?
  !!current_user
end
```

The logged_in? action just checks whether or not there is a current_user. Two exclamations turns it into a boolean. *True or false, we have a user logged in?*

The final action determines what happens if a user is not(**!**) logged in.

```ruby
def require_user
  if !logged_in?
    flash[:danger] = "You must log in to perform that action"
    redirect_to root_path
  end
```

We know the **session** will hold the user info and we already have a signup button. Lets add a conditional to the navigation partial that gives ***login/signup*** when the user **session** is empty(or, user logged out), and a ***logout*** when the user **session** is full (or, user logged in):

```
<ul class="nav navbar-nav navbar-right">
  <% if logged_in? %>
    <li><%= link_to 'Logout', logout_path, method: :delete %></li>
    <li class="dropdown">
      <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button"
      <ul class="dropdown-menu">
        <li><a href="#">Action</a></li>
        <li><a href="#">Another action</a></li>
        <li><a href="#">Something else here</a></li>
        <li role="separator" class="divider"></li>
        <li><a href="#">Separated link</a></li>
      </ul>
    </li>
  <% else %>
    <li><%= link_to 'Login', login_path %></li>
    <li><%= link_to 'Sign-up', signup_path %></li>
  <% end %>
```

Give it a go, make sure the nav status changes between logging in and out of the development app.

Before we move to set more user authentication, lets extract the redundancies from **edit**, **update** and **show** in the UserController, and add:

```
before_action :set_user, only: [:edit, :update, :show]
```

And the private action:

```
def set_user
  @user = User.find(params[:id])
end
```

We implemented the authentication actions in ApplicationController. Now its time for...

Restricting actions from the browser

With the conditional statement we added to the navigation, lets add some access for the logged in user:

```
<% if logged_in? %>
  <li><%= link_to 'Logout', logout_path, method: :delete %></li>
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopu
      ="false">Your Profile <span class="caret"></span></a>
    <ul class="dropdown-menu">
      <li><%= link_to "Edit your profile", edit_user_path(current_user) %></li>
      <li><%= link_to "View your profile", user_path(current_user) %><li>
      <li><a href="#">Something else here</a></li>
      <li role="separator" class="divider"></li>
      <li><a href="#">Separated link</a></li>
    </ul>
  </li>
<% else %>
  <li><%= link_to 'Login', login_path %></li>
  <li><%= link_to 'Sign-up', signup_path %></li>
<% end %>
```

Edit the navbar with the links you want and use the logged_in? method to define access.

Use the **!**logged_in? to create a conditional that removes the large signup button when the user is logged in:

```
<% if !logged_in? %>
  <%= link_to "Sign up", signup_path, clas
    >
<% end %>
```

Or, redirect the home action to the articles index:

```
def home
  redirect_to articles_path if logged_in?
end
```

Not both, the redirect will take precedence.

Now lets go through the views and find where we want to add restrictions...

The first one we come across, the articles index partial, has **edit** and **delete** buttons at the bottom of the page. While the user can view every Article object, we only want them to destroy or edit the articles they created. We manage this by determining if the user is logged in AND if that user is the same user who's data is saved in the session hash:

```
<% if logged_in? && current_user == article.user %>
  <div class="article-actions">
    <%= link_to "Edit", edit_article_path(article), class: "btn btn-xs btn-primary" %>
    <%= link_to "Delete", article_path(article), method: :delete,
                data: { confirm: "Are you sure?" },
                class: "btn btn-xs btn-danger" %>
  </div>
<% end %>
```

Now as you view all the articles, only your articles are available to edit. We can do the same for the show views:

```
<div class="article-actions">
  <%= link_to "View all articles", articles_path, class: "btn btn-xs btn-primary" %>
  <% if logged_in? && current_user == @article.user %>
    <%= link_to "Edit this article", edit_article_path(@article), class: "btn btn-xs btn-primary"
      %>
    <%= link_to "Delete this article", article_path(@article), method: :delete,
                data: { confirm: "Are you sure?" },
                class: "btn btn-xs btn-danger" %>
  <% end %>
```

Notice there is no block in the show view and therefor we use @article as it is defined in the show action.

As we navigate through the app now, we only have access to the elements we created. We can't navigate to a point where we can edit or delete another users creation. For example, if we view another users profile page, there isn't an edit button... But what stops us from adding /edit to the URL and editing data? Nothing. Adding restrictions that prevent us from exploiting a URL must be done at the controller level. To prevent somebody from gaining access by typing articles/6/edit into the address bar, must be done in the ArticleController. Likewise, users data must be protected in the UserController.

We can use the require_user action/method as a *before_action* in the controller, which displays a message if a user is not logged in and redirects to the root path, before allowing the user to pass through that controller.

```
class UsersController < ApplicationController
  before_action :set_user, only: [:edit, :update, :show]
```

Its important to note that *before_action*'s are performed *in order*. In the case of *set_user*, we need this to set the instance variables for...

...edit, update, show, etc. Only then can we set required_user.

```
class ArticlesController < ApplicationController
  before_action :set_article, only: [:edit, :update, :show, :destroy]
```

In this case, we want to add restrictions to almost all of our controller actions. In fact, only the actions that display articles will not require the user to be logged in. Its easier to add the exceptions:

```
class ArticlesController < ApplicationController
  before_action :set_article, only: [:edit, :update, :show, :destroy]
  before_action :require_user, except: [:index, :show]
```

Now the user and the users articles are protected... kind of... A visitor to the app has no access, but the requirement only says you have to be logged in- it doesn't say to which user account. So now, any logged in user can manipulate any other users data. This is a step in the right direction, but lets add one more level of security.

Since we established a user with require_user, lets add a controller-specific private action to make sure that user is equal to the @article.user:

```
def require_same_user
  if current_user != @article.user
    flash[:danger] = "You can only edit or delete your own articles"
    redirect_to root_path
  end
end
```

Then, we use this new action and restrict based on the user that created the instance.

```
class ArticlesController < ApplicationController
  before_action :set_article, only: [:edit, :update, :show, :destroy]
  before_action :require_user, except: [:index, :show]
  before_action :require_same_user, only: [:edit, :update, :destroy]
```

The restrictions for UserController follow the same guidelines, with a redirect back to the current users profile:

```
def require_same_user
  if current_user != @user
    flash[:danger] = "You can only edit your own profile"
    redirect_to user_path(current_user)
  end
end
```

```
class UsersController < ApplicationController
  before_action :set_user, only: [:edit, :update, :show]
  before_action :require_same_user, only: [:edit, :update]
```

One thing we need to update is the ArticlesController create action. The *before_action* has already required a user for the controller and the create action needs to use the same data:

```
def create
  #render plain: params[:article].inspect
  #This displays the params hash data in the browser
  @article = Article.new(article_params)
  @article.user = current_user
  #pulls the specified attributes from params with pri
  if @article.save
    flash[:info] = "Article was successfully created"
```

Currently, when a new user signs up they immediately have to login to gain access. We can skip the login after signup by saving the signup data in the users session:

Now, after signup the user is logged in and taken to their profile page.

```
def create
  @user = User.new(user_params)
  if @user.save
    sessions[:user_id] = @user.id
    flash[:info] = "Welcome to the conversation, #{@user.username}"
    redirect_to user_path(@user)
  else
    render :new
  end
end
```

Admin User functionality

There are three ways to implement admin status:

1. **Permission table with roles and associated permissions**

    For apps with multiple layers of complex actions and user roles, where each action and/or user requires different permissions. Database heavy.

2. **A column in users table specifying role, for example if a user is an admin or a moderator**

    When there is a choice between just a few roles, each role can be represented by a string and added as an attribute.

3. **A boolean admin column which starts all users as false for that column, unless you set a user to admin**

    Simplest way to set admin. This is the method we will set up below.

An admin is a user(s) with special permissions. Typically the admin can delete and edit other users data and help manage an app from within.

First lets create a new migration file:

rails generate migration add_admin_to_users

Then add a new column...

```
class AddAdminToUsers < ActiveRecord::Migration
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

...and run rake.

```
2.3.0 :002 > user = User.last
  User Load (0.4ms)  SELECT  "users".* FROM "users"  ORDER BY "users"."id" DESC LIMIT 1
 => #<User id: 16, username: "captcrunch", email: "capt@crunch.com", created_at: "2016-10-03
 19:53:38", updated_at: "2016-10-03 19:53:38", password_digest: "$2a$10$fBaL/MCahfZ97WzALDud
u.FvgKVA9jfJzw6Ogmoq1uP...", admin: false>
2.3.0 :003 >
```

If we jump to the console and pull up the last user:

We can see the default setting for admin is false.

```
2.3.0 :011 > x = User.find(7)
  User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT 1  [["id",
7]]
 => #<User id: 7, username: "b_random", email: "brandon.brigance@gmail.com", created_at: "20
16-09-22 03:20:05", updated_at: "2016-09-22 03:20:05", password_digest: "$2a$10$xiSf9jL6Vf88
lt7rdxjy3ep2E40PNsHfmuYByYRqyQf...", admin: false>
2.3.0 :012 > x.toggle!(:admin)
   (0.3ms)  begin transaction
  SQL (0.6ms)  UPDATE "users" SET "admin" = ?, "updated_at" = ? WHERE "users"."id" = ?  [["a
dmin", "t"], ["updated_at", "2016-10-04 00:16:58.357636"], ["id", 7]]
   (13.2ms)  commit transaction
 => true
2.3.0 :013 > x
 => #<User id: 7, username: "b_random", email: "brandon.brigance@gmail.com", created_at: "20
16-09-22 03:20:05", updated_at: "2016-10-04 00:16:58", password_digest: "$2a$10$xiSf9jL6Vf88
lt7rdxjy3ep2E40PNsHfmuYByYRqyQf...", admin: true>
2.3.0 :014 >
```

To set a user to admin from the console use *user.toggle!(:admin)*:

Now that we have an admin set, lets add the admins special permissions.

Remember how the ArticleController uses require_same_user as a *before_action* to make sure that a user trying to edit an article, is the same user that created the article?

```
before_action :require_same_user,
```

+

```
def require_same_user
  if current_user != @article.user
    flash[:danger] = "You can only edit or delete your own articles"
    redirect_to root_path
  end
end
```

All we have to do is adjust the conditional to not exclude the admin:

```
def require_same_user
  if current_user != @article.user && !current_user.admin?
    flash[:danger] = "You can only edit or delete your own articles"
    redirect_to root_path
  end
end
```

Not much to it- Don't forget to change permission conditionals in views/articles:

```
<% if logged_in? && (current_user == article.user || current_user.admin?) %>
  <div class="article-actions">
```

Example: articles/_article.html.erb

Time to define admin conditionals in UserController.

We never created a destroy action for the UsersController, so lets do that first:

```
def destroy
  @user = User.find(params[:id])
  @user.destroy
  flash[:danger] = "User and all articles created by the user have been deleted"
  redirect_to users_path
end
```

```
before_action :require_same_user, only: [:edit, :update, :destroy]
before_action :require_admin, only: [:destroy]
```

Add the destroy action to require_same_user and the new require_admin to the *before_action*

This will require admin status to delete a user.

```
def require_admin
  if logged_in? && !current_user.admin?
    flash[:danger] = "Only admin users can perform that action"
    redirect_to root_path
```

```
class User < ActiveRecord::Base
  has_many :articles, dependent: :destroy
  before_save { self.email = email.downcase }
  #saves email in downcase
  validates :username, presence: true,
            uniqueness: { case_sensitive: false },
            #ensures unique username + no case sense (joe = Joe
            length: { minimum: 3, maximum: 25 }
  VALID_EMAIL_REGEX= /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 105 },
            uniqueness: { case_sensitive: false },
            format: { with: VALID_EMAIL_REGEX}
            #using regex for proper email address format
  has_secure_password
end
```

But how do we delete all the articles when the user is destroyed? Just add a dependency to the User model/articles association:

Now the articles die with the user.

All we need is a destroy link now. Lets add one to the users index page that restricts use to only the admin:

```
<h1 align='center'>Users</h1>


<div align="center">
  <%= will_paginate %>
  <% @users.each do |user| %>
    <ul class="listing">
      <div class="row">
        <div class="well col-md-4 col-md-offset-4">
          <li><%= link_to gravatar_for(user), user_path(user) %></li>
          <!--gravatar image as link to users show page-->
          <li class="article-title user-title">
            <%= link_to user.username, user_path(user) %>
          </li>
          <li><small><%= pluralize(user.articles.count, "article") if user.articles %></small
            ></li>
          <!--pluralize displays the number and pluralizes the keyword when necessary-->
          <% if logged_in? && current_user.admin? %>
            <li><%= link_to "Delete this user", user_path(user), method: :delete, data: {
              confirm: "Are you sure you want to delete the user and all their articles?" } %
              ></li>
          <% end %>
        </div>
      </div>
    </ul>
  <% end %>
  <%= will_paginate %> <!--Adds per page navigation-->
</div>
```

Log in as the admin and test it out.

This works for our development app because we know how to access the db from the CLI. But once we push the app to production, how do we set an admin user?

In the production app, create a few users and a few articles for each user through the browser. Then log into the profile that you want to set as the admin user.

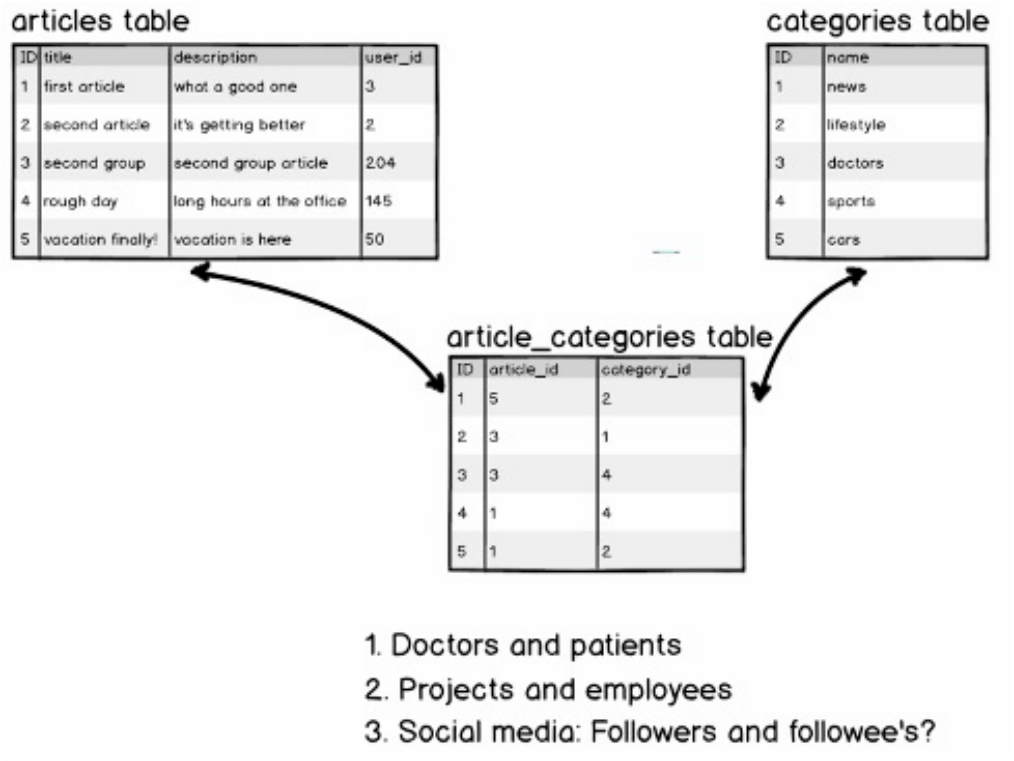
From the CLI, run:

heroku run rails console

Now we can access the rails console in production and change the admin status for our user with the *toggle!* method. The rest of the views permissions were updated to production with the last rake.

Refresh the production app and make sure it works.

The basics of our rails app is complete. Lets move onto some more advanced concepts.



This ERD (entity relationship diagram) shows how we can use a **many to many** association and add categories to the app. Both Article and Category have a table, and then the ArticleCategories table defines the relationships between them. An instance of Article can be associated with multiple categories and vice versa, hence **many to many**.

1-3 offers examples of **many to many** associations.

As we build the tables/relationships above, we will also begin to test the implementation & build automated tests:

- **Unit Tests**: For models & individual units of the application (like validations) are working.
- **Functional Tests**: Controllers/functions are working. I.e. before_action preventing a non-logged in user from performing an action.
- **Integration Tests**: Full features/ start to finish of a business process. I.e. a user signs up or checking out.

TDD, or **Test Driven Development** is the method of writing tests that fail for desired functionality, then write the code which adds the functionality, and finally ensure that the test passes.

Testing support is woven into Rails. Even as you create models and controllers, Rails is always working to create skeleton test code in the background.

http://guides.rubyonrails.org/testing.html breaks it down:

```
$ ls -F test
controllers/    helpers/        mailers/        test_helper.rb
fixtures/       integration/    models/
```

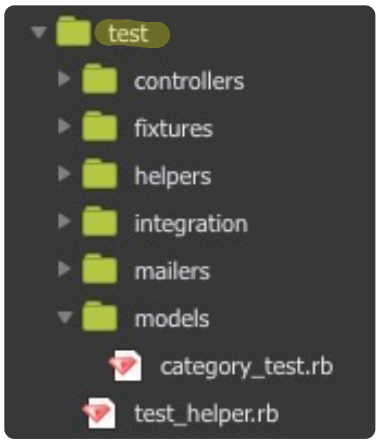
The `models` directory is meant to hold tests for your models, the `controllers` directory is meant to hold tests for your controllers and the `integration` directory is meant to hold tests that involve any number of controllers interacting. There is also a directory for testing your mailers and one for testing view helpers.

It also tells us that Rails holds test configurations in a method called *test_helper.rb*, which is required in the test files.

Follow the link to read in depth. In this tutorial we will build the tests manually.

All tests will be added to the test directory, under the proper sub-directory.

- Lets add a file called test/model/category_test.rb




```
require 'test_helper'

class CategoryTest < ActiveSupport::TestCase



end
```

Here is our new category_test.rb test file:

As with every test, we require the *test_helper.rb* method.

The class defines the *test case* and inherits from ActiveSupport and TestCase. This acts as our test template.

setup is an action that always runs first- before the test. Here we use it to define an instance variable, @category, that we can then use ***in*** the test.

An assertion evaluates an object or expression and expects results. In other words, its what we are testing for.

The test method allows readable blocks in place of actions. This test block is the same as:

```
def valid_category
    assert @category.valid?
end
```

```ruby
require 'test_helper'

class CategoryTest < ActiveSupport::TestCase

  def setup
    @category = Category.new(name: "sports")
  end

  test "Category should be valid" do
    assert @category.valid?
  end


end
```

Here we are testing whether we can initiate a new category instance variable and whether it is valid. Run the test rake test:

```
b_random_sb:~/workspace (master) $ rake test
Running via Spring preloader in process 131377
Run options: --seed 857

# Running:

E

Finished in 0.007054s, 141.7546 runs/s, 0.0000 assertions/s.

  1) Error:
CategoryTest#test_Category_should_be_valid:
NameError: uninitialized constant CategoryTest::Category
    test/models/category_test.rb:6:in `setup'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

Basically, the error is saying we don't have a Category model and the test failed. Our goal is to make the test pass... We need a Category model:

Save the new model and run rake test again:

```ruby
class Category < ActiveRecord::Base

end
```

The test failed again, but for a different reason. It could not find a categories table:

```
  1) Error:
CategoryTest#test_Category_should_be_valid:
ActiveRecord::StatementInvalid: Could not find table 'categories'
    test/models/category_test.rb:6:in `setup'
```

Lets create a migration and add an attribute to the categories table:

```
class CreateCategories < ActiveRecord::Migration
  def change
    create_table :categories do |t|
      t.string :name
      t.timestamps
    end
  end
end
```

Check the schema for errors in our new table.

Run the category test again...

```
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

...and it passes.

Now we know we can initiate a Category instance variable and it will be valid.

When you run a test, the test never hits the development db. Everything takes place in a separate test db and data is pulled from the test file. The test db is wiped after each test.

You can create the instance in the CLI to confirm:

```
2.3.0 :001 > c = Category.new(name: "sports")
 => #<Category id: nil, name: "sports", created_at: nil, updated_at: nil>
```

The test above didn't really do a whole lot when you consider all the things the model provides. Lets keep building and test presence, uniqueness and length validations.

For presence of **name** attribute:

Here we test that the **name** attribute is equal to an empty string. This should validate false, because whether a **name** is present or not, it still doesn't equal a blank string. But, we still want the test to assert true- with *assert_not* the test returns true, even with a false validation.

```
test "name should be present" do
  @category.name = " "
  assert_not @category.valid?
end
```

```
  1) Failure:
CategoryTest#test_name_should_be_present [/home/ubun
ry_test.rb:15]:
Expected true to be nil or false

2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

Because there are no validations set in the model, anything will validate true, which gives us a failing test.

Lets add validations to models/category.rb:

```ruby
class Category < ActiveRecord::Base
  validates :name, presence: true
end
```

If we run the test again, it should pass.

To test for **name** uniqueness

- -@category is saved to the test db with its **name** attribute
- -Create a new instance with the same **name** attribute
- -Test if the new instance is not valid

```ruby
test "name should be unique" do
  @category.save
  category2 = Category.new(name: "sports")
  assert_not category2.valid?
end
```

Again, when we run this test we get an "Expected true to be nil or false" failure. There is no validation to say we can't have more than one of the same **name** attribute.

After we add the validation, the test will pass:

```ruby
class Category < ActiveRecord::Base
  validates :name, presence: true
  validates :name, uniqueness: true
end
```

```ruby
test "name should not be too long" do
  @category.name = "a" * 26
  assert_not @category.valid?
end

test "name should not be too short" do
  @category.name = "aa"
  assert_not @category.valid?
end
```

Because there aren't any validations for length, both of these names *are* valid and the test fails.

Once we add min and max validations to the model that limit a **name** to 3-25 characters, these tests will pass.

Now the Category table is established and we can create instances from the console. We know the model is working correctly because we ran a series of tests. This is more efficient than having to build out the rest of the MVC just to make sure one part is working like it should. Now, lets turn towards the browser and get the UI working.

Browser requests come in through routes to the controller. Lets start with test/controllers/**categories_controller_test.rb**.

The controller test file has a different sub-class then the model test:

```
require 'test_helper'

class CategoriesControllerTest < ActionController::TestCase

end
```

Think about the actions you want the controller to perform... An index page to list categories, a show page to display the articles that match a specific category, a way to create/delete categories, etc. Start by creating the block for each test:

```
class CategoriesControllerTest < ActionController::TestCase

  test "should get categories index" do

  end

  test "should get new" do

  end

  test "should get show" do

  end

end
```

```
test "should get categories index" do
  get :index
  assert_response :success
end

test "should get new" do
  get :new
  assert_response :success
end

test "should get show" do
  get :show
  assert_response :success
end
```

Each of these tests is the same, with the exception of the particular route.

The test returns successful when it finds each of the actions in the CategoriesController.

Run the test and you get the same error for each:

```
  1) Error:
CategoriesControllerTest#test_should_get_categories_index:
RuntimeError: @controller is nil: make sure you set it in your test
    test/controllers/categories_controller_test.rb:6:in `block in <
```

Create the CategoriesController and run the test again:

```
  1) Error:
CategoriesControllerTest#test_should_get_new:
ActionController::UrlGenerationError: No route matches {:action=>"ne
    test/controllers/categories_controller_test.rb:11:in `block in <
```

"No route matches" x3. This is beginning to look like the Rails browser messages from the the first MVC build... Which is kind of the point.

Lets create the routes (no need for a destroy action for categories)...

```
resources :categories, except: [:destroy]
```

...and try the test again...

Now the error says we are missing the specified actions:

```
test_should_get_categories_index:
nNotFound: The action 'index' could not be found for CategoriesController
gories_controller_test.rb:6:in `block in <class:CategoriesControllerTest>
```

You know how to fix this problem....

```
class CategoriesController < ApplicationController
  def index
  end

  def show
  end

  def new
  end
end
```

And can you guess what the next error will be?

```
  1) Error:
CategoriesControllerTest#test_should_get_new:
ActionView::MissingTemplate: Missing template categories/new, application/new with {:locale=>[:en]
, :formats=>[:html], :variants=>[], :handlers=>[:erb, :builder, :raw, :ruby, :coffee, :jbuilder]}.
 Searched in:
  * "/home/ubuntu/workspace/app/views"

    test/controllers/categories_controller_test.rb:11:in `block in <class:CategoriesControllerTest
>'


  2) Error:
CategoriesControllerTest#test_should_get_show:
ActionController::UrlGenerationError: No route matches {:action=>"show", :controller=>"categories"
}
    test/controllers/categories_controller_test.rb:16:in `block in <class:CategoriesControllerTest
>'


  3) Error:
CategoriesControllerTest#test_should_get_categories_index:
ActionView::MissingTemplate: Missing template categories/index, application/index with {:locale=>[
:en], :formats=>[:html], :variants=>[], :handlers=>[:erb, :builder, :raw, :ruby, :coffee, :jbuilde
```

Missing template, of course. Then why is the **show** test still saying "*no route matches*"? Remember, the **show** route requires an id, for example: users/7. Lets set the views for all three actions and then we can fix the **show** route.

Add views/categories/index.html.erb and new.html.erb, then run the test again. The only error we should see is about the **show** route.

To correct the show route we need a **setup** action to define an instance variable, much like we did with the model test. This time however, we use create instead of new because we want the instance to hit the test db so it can be referenced by the **show** action:

```
def setup
  @category = Category.create(name: "sports")
end
```

Then modify the show test, defining id:

```
test "should get show" do
  get(:show, {'id' => @category.id})
  assert_response :success
end
```

Run the test one last time- there should be no errors!

Implement create Category feature using integration test

Under test/integration create a new file called create_categories_test.rb. Here we want to immolate user & MVC behavior when creating a new category:

Notice, again, our class pulls from a different sub-class

```
require 'test_helper'

class CreateCategoriesTest < ActionDispatch::IntegrationTest

  test "get new category form and create category" do
    get new_category_path
  end

end
```

You shouldn't be surprised when this test passes- The new path has already been built...

```
test "get new category form and create category" do
  get new_category_path
  assert_template 'categories/new'
end
```

Same with this template test. We already built this out, so this test should pass too.

We have a route test and a template test, but how do we simulate a create test? (**#see image notes for explanation**)

```ruby
require 'test_helper'

class CreateCategoriesTest < ActionDispatch::IntegrationTest

  test "get new category form and create category" do
    get new_category_path
    #following the new category path
    assert_template 'categories/new'
    #get the new form
    assert_difference 'Category.count', 1 do
    #verifies that the result of evaluating Category.count changes by 1 after calling the block below
      post_via_redirect categories_path, category: {name: "sports"}
      #post category: to categories_path
    end
    assert_template 'categories/index'
    #is user sent to categories/index page? and...
    assert_match "sports", response.body
    #...verify a matching category called sports in the body
  end
end
```

Run through the script a few times to make sure you have a good understanding of what is happening. We have four assert methods that could return errors.

Now run the test...

```
  1) Error:
CreateCategoriesTest#test_get_new_category_form_and_create_categor
y:
AbstractController::ActionNotFound: The action 'create' could not
be found for CategoriesController
```

The only thing that throws an error is POST test (*post_via_redirect*), which *is the create action*. Add create to CategoriesController and run the test again...

```
ActionView::MissingTemplate: Missing template categories/create, a
pplication/create with {:locale=>[:en], :formats=>[:html], :varian
```

Now the error says there is no create template- but create gets **params** data passed from the new template form. Lets add the form:

```erb
<h1 align="center">Create new categories</h1>

<%= render "share/errors", {obj: @category} %>

<%= form_for(@category, :html => {class: "form-horizontal", role:
  "form"}) do |f| %>
  <div class="form-group">
    <div class="control-label col-sm-2">
      <%= f.label :name %>
    </div>
    <div class="col-sm-8">
      <%= f.text_field :name, class: "form-control", placeholder:
        "Enter category name", autofocus: true %>
    </div>
  </div>

  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <%= f.submit class: "btn btn-primary btn-lg" %>
    </div>
  </div>
<% end %>
<div class="col-xs-4 col-xs-offset-4">
  [<%= link_to "Cancel request and return to articles listing",
    articles_path %>]
</div>
```

Keeping with the style we've been using, copy and paste the users/_form and make the necessary changes. As long as we define @category in the new action the _errors partial should render with no problems.

```ruby
def new
  @category = Category.new
end
```

To run only one test, just provide the path: rake test test/integration/create_categories_test.rb

We have the form and the submit button, but the create action is still empty; Running the test here will give the same error. Its time to build the create action:

```
def create
  @category = Category.new(category_params)
  if @category.save
    flash[:info] = "Category successfully created"
    redirect_to categories_path
  else
    render :new
  end
end

private

def category_params
  params.require(:category).permit(:name)
end
```

Run the test again:

```
  1) Failure:
CreateCategoriesTest#test_get_new_category_form_and_c
reate_category [/home/ubuntu/workspace/test/integrati
on/create_categories_test.rb:17]:
Expected /sports/ to match "<!DOCTYPE html>\n<html>\n
<head>\n  <title>Workspace</title>\n  <link rel=\"sty
lesheet\" media=\"all\" href=\"/assets/application-a6
```

Something new. A failure occurs in this case when the actions work, but the outcome is different than expected. Remember our test was looking for a new category called "sports", but couldn't find it. If we go through each test, we can get a better understanding of what created the **failure:**

```
test "get new category form and create category" do
  get new_category_path
  assert_template 'categories/new'
  assert_difference 'Category.count', 1 do
    post_via_redirect categories_path, category: {name: "sports"}
  end
  assert_template 'categories/index'
  assert_match "sports", response.body
end
```

- Pass: We have a new template/form
- Pass: The count moves by +1
- Pass: POST/create action works- posts "sports"
- Pass: We have an index page
- Fail: Does not match "sports" in the body of said index page

```
<h2 align="center">All Categories</h2>

<% @categories.each do |category| %>
  <div class="listing">
    <div class="row">
      <div class="well col-md-4 col-md-offset-4">
        <li class="article-title">
          <%= link_to "#{category.name}", category_path(category) %>
        </li>
      </div>
    </div>
  </div>
<% end %>
```

We have an index page, but there isn't anything there. Lets build it out to display all the categories.

Don't forget to define the instance variable

```
def index
  @categories = Category.all
end
```

Now all the tests will pass. This is TDD! Create s couple of categories in the browser to watch all that testing pay off. Check that the errors work, etc.

Now lets create a scenario where invalid data is entered. In this scenario we do not want the invalid category created; we want to make sure that the errors display, and that this failure re-renders the new template as determined by the create function conditional.

Just a few changes will do the trick:

```
assert_template 'categories/new'
```

- look for the new template

```
assert_no_difference 'Category.count' do
```

- There should not be additional entries when...

```
post_via_redirect categories_path, category: {name: " "}
```

- ...the name attribute is left empty

```
assert_template 'categories/new'
```

- do we end up back at the new template with...

```
assert_select 'h2.panel-title'
assert_select 'div.panel-body'
```

- ...errors present in these two divs/classes

This should not return errors or failures.

You might notice we used the same method for POST, *post_via_redirect*. Because we are not intending for this test to POST, we want to be syntactically correct, and just use the method *post*:

```
post categories_path
```

We want to test that each category is listed, and that each listing is a link to the show page. This is still a "**create category test**", so the class does not change:

```
class CreateCategoriesTest < ActionDispatch::IntegrationTest

  def setup
    @category = Category.create(name: "sports")
    @category2 = Category.create(name: "programming")
  end

  test "should show categories listing" do
    get categories_path
    assert_template 'categories/index'
    assert_select "a[href=?]", category_path(@category), text: @category.name
    assert_select "a[href=?]", category_path(@category2), text: @category2.name
    #assert that a[href] link leads to the show page of each instance and displays the name
  end

end
```

Add this to **integration/list_categories_test.rb** with the same class

- Setup: create the new categories instances, "sports" and "programming"
- Are they present on the index page
- Is there a show page link

If we run just this test, it passes. If the entire test suite is run, there is a single failure...

```
  1) Failure:
CreateCategoriesTest#test_get_new_category_form_and_create
ce/test/integration/create_categories_test.rb:8]:
"Category.count" didn't change by 1.
Expected: 3
  Actual: 2
```

Why? Because it it has the same class name as the last test(**create_categories_test.rb**), and when all assertions are run together, the two "sports" instances get lumped together. If we change this most recent instance to something else, like "books", the test db is hit with three separate instances. Or just change the class name, so they don't run as the same test.

```
@category = Category.create(name: "books")
```

```
<h2 align="center">All Categories</h2>


<div align="center">
  <% will_paginate %>
  <% @categories.each do |category| %>
    <div class="listing">
      <div class="row">
        <div class="well col-md-4 col-md-offset-4">
          <li class="article-title">
            <%= link_to "#{category.name}", category_path(category) %>
            <!--category_path == show page-->
          </li>
        </div>
      </div>
    </div>
  <% end %>
  <% will_paginate %>
</div>
```

```
def index
  @categories = Category.paginate(page: params[:page], per_page: 5)
```

Go ahead and add pagination to the categories#index page, using the standard pagination implementation:

**ruby tags missing "="

With any changes, run the test to ensure everything still passes.

Here is what the test looks like with the same instance of category, but updated class name.

This should run without error as well.

```
require 'test_helper'

class ListCategoriesTest < ActionDispatch::IntegrationTest

  def setup
    @category = Category.create(name: "sports")
    @category2 = Category.create(name: "programming")
  end

  test "should show categories listing" do
    get categories_path
    assert_template 'categories/index'
    assert_select "a[href=?]", category_path(@category), text: @category.name
    assert_select "a[href=?]", category_path(@category2), text: @category2.name
    #assert that a[href] link leads to the show page of each instance and displa
  end

end
```

Further Explanation: Due to the same class name of CreateCategoriesTest, an instance variable (object) initiated in 1 test lasted the lifecycle of the object which spanned multiple files (since same class), thus causing the conflict. Correcting the class name removed the list_categories_test.rb file from being part of the CreateCategoriesTest class and the @category intance variable was no longer spanning that file (and it's tests) and promptly ended when the create_categories_test.rb file completed.

Admin User Requirements Testing

As of now, anybody can create a new category. We want to limit this ability to admins. This restriction takes place in the controller, so lets take a look at the controller tests, **categories_controller_test.rb**...

- Assert that no instance is added to the test db, count does not change
- verify redirect to the index page

```
test "should redirect create when admin not logged in" do
  assert_no_difference 'Category.count' do
    post :create, category: {name: "sports" }
  end
  assert_redirected_to categories_path
end
```

For test use rake test test/controllers/categories_controller_test.rb

```
  1) Failure:
CategoriesControllerTest#test_should_redirect_create
tu/workspace/test/controllers/categories_controller_
Expected response to be a <redirect>, but was <200>
```

The test stopped when the instance **did** hit the db, and we didn't get a redirect; Hence the failure message. 200 is the html code for success.

We need to build a *require_admin* method for CategoreisController like we did for the other controllers. Then use it in a *before_action* to prevent regular users from gaining access and creating new instances.

```
before_action :require_admin, except: [:index, :show]
```

```
def require_admin
  if !logged_in? || (logged_in? && !current_user.admin?)
    # if not logged in || logged in but not the admin
    flash[:danger] = "Only admin can perform that action"
    redirect_to categories_path
  end
end
```

Now when the test is run, the POST attempt is rejected by the *before_action* which refers to the *require_admin* method. This means there is "no count difference" and the test follows through to redirect.

Lets run just this test again, rake test test/controllers/categories_controller_test.rb ...

```
  1) Failure:
CategoriesControllerTest#test_should_get_new [/home/
egories_controller_test.rb:15]:
Expected response to be a <success>, but was <302>
```

Uh-oh, where did this come from? The failure gives you call the info. First the class and the individual test name, then the file name followed by the actual error. A test from earlier, "**test_should_get_new**", is not successful. 302 is another html code that means there was a redirect. Lets look at this test:

```
test "should get new" do
  get :new
  assert_response :success
end
```

Its asserting a successful **new** action- The error is caused by the *before_action* we just added, which restricted access to all but the index and show actions, from anybody who is not an admin. Now when this test is run, the before method is preventing success. To fix the issue, we need to create an admin user in the CategoriesControllerTest and apply that user to this failed test:

```
@user = User.create(username: "john", email: "john@example.com", password:
  "password", admin: true)
```

```
test "should get new" do
  session[:user_id] = @user.id
  get :new
  assert_response :success
end
```

Because this is a controller test, we can just create a **sessions** hash to offer the users id- run it again for success!

Now attempt the entire test suite?

```
  1) Failure:
CreateCategoriesTest#test_get_new_category_form_and_cre
untu/workspace/test/integration/create_categories_test.
expecting <"categories/new"> but rendering with <[]>


  2) Failure:
CreateCategoriesTest#test_invalid_category_submission_r
me/ubuntu/workspace/test/integration/create_categories_
expecting <"categories/new"> but rendering with <[]>
```

Some of our previous tests need to be updated with an admin user as well. We can create a user in the **setup** action of CreateCategoryTest like we did above, but the tests in this class aren't run through the controller and therefor do not have access to the sessions hash. We need to add a user another way...

We can create a method that the CreateCategoryTest tests can call upon to provide a user. Under test/test_helper.rb is where we add methods that are accessible to all the test directories.:

```ruby
class ActiveSupport::TestCase
  # Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
  fixtures :all

  # Add more helper methods to be used by all tests here...
end
```

test/test_helper.rb

After the user setup is created we can simulate a user signing in with a new method

```ruby
  # Add more helper methods to be used by all tests here...
  def sign_in_as(user, password)
    post login_path, session: {email: user.email, password: password}
  end
end
```

```ruby
test "get new category form and create category" do
  sign_in_as(@user, "password")
  get new_category_path
  assert_template 'categories/new'
  assert_difference 'Category.count', 1 do
    post_via_redirect categories_path, category: {name: "sports"
  end
  assert_template 'categories/index'
  assert_match "sports", response.body
end

test "invalid category submission results in failure" do
  sign_in_as(@user, "password")
```

Just plug the method into both tests under CreateCategoryTest class, and run the tests again...

...No errors/failures!

```erb
<% if logged_in? %>
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria
      -expanded="false">Explore <span class="caret"></span></a>
    <ul class="dropdown-menu">
      <li><%= link_to 'Create an article', new_article_path %></li>
      <li role="separator" class="divider"></li>
      <li><%= link_to 'All Categories', categories_path %></li>
      <% Category.all.each do |category| %>
        <li><%= link_to "#{category.name}", category_path(category) %></li>
      <% end %>
      <% if logged_in? && current_user.admin? %>
        <li role="separator" class="divider"></li>
        <li><%= link_to "Create a category", new_category_path %></li>
      <% end %>
    </ul>
  </li>
<% else %>
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria
      -expanded="false">Explore <span class="caret"></span></a>
    <ul class="dropdown-menu">
      <li><%= link_to 'All Categories', categories_path %></li>
      <% Category.all.each do |category| %>
        <li><%= link_to "#{category.name}", category_path(category) %></li>
      <% end %>
    </ul>
  </li>
<% end %>
```

The only thing we are missing from the categories MVC is some UI...

Lets add that now. When the site is being viewed as an unregistered user, only the categories should display. Users can see the categories and "create an article", while admin adds "create a category":

ARTICLE AND CATEGORY ASSOCIATION

Create a new table migration:

rails generate migration create_article_categories

integer :article_id
integer :category_id

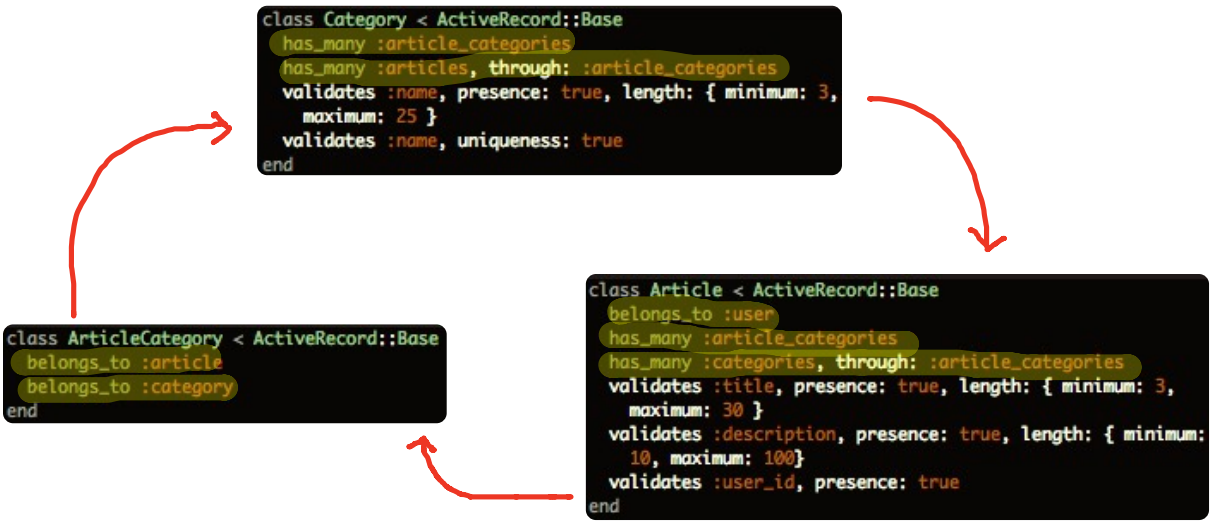
And follow it up with the model:

models/article_category.rb

Now we have enough to test the db table in the rails console (don't forget to **reload!**)

```
 => #<ActiveRecord::Relation []>
2.3.0 :020 > ArticleCategory
 => ArticleCategory(id: integer, article_id: integer, category_id: integer)
2.3.0 :021 > exit
```

Time to associate the models.



Take notice that we utilize the **through:** method.

Now that we have all these relationships between db models, how do we use them? Lets take a look in the console:

- If we pull up an article with *article = Article.last*, then we can see the associated categories with *articles.categories*.

- Similarly, *category = Category.first* shows us the first category. If we want this category added to the article from above, we can say *category.articles << article*. Basically saying that the last article instance is one of the articles associated with the first category.

  Now when we look up *ArticleCategory.all*, we see the article and the categories that represent it:

```
[#<ArticleCategory id: 1, article_id: 31, category_id: 1>]>
```

article.categories and *category.articles* displays two sides of the same coin.

- Before we added an article to a category; This time lets add a category to an article... if *c = Category.last* and *a = Article.second*, then *a.categories << c*.

  Grab a different Category instance, maybe c = Category.third, and add it to *a* with *a.categories << c*.

  Another way to do this same thing is *a.categories << Category.find(3)*.

Its time to apply this to the UI and allow users to set a category when they create an article.

A new article can be made at **articles/new.html.erb** via **_form.html.erb**. Lets add a category selector to this form:

Duplicate the submit block and turn it into a row. Paste it into the form directly above the submit block:

```
<div class="form-group">
  <div class="row">
    <div class="col-sm-offset-2 col-sm-10">
      <%= f.submit class: "btn btn-primary btn-lg" %>
    </div>
  </div>
</div>
```

```
<div class="form-group">
  <div class="row">
    <div class="col-sm-offset-2 col-sm-8">
      <%= f.collection_check_boxes :category_ids, Category.all, :id, :name do |cb| %>
        <% cb.label(class: "checkbox-inline input_checkbox") {cb.check_box(class:
          "checkbox") + cb.text} %>
      <% end %>
      <!--creates checkboxes-->
    </div>
  </div>
</div>
```

This is the special block of code that creates check boxes.

To better understand, read http://apidock.com/rails/v4.0.2/ActionView/Helpers/FormOptionsHelper/collection_check_boxes

Now each category is displayed alongside a checkbox, just above the submit button. As with the rest of the form data, the category_ids are sent via **params**. Just need to configure the ArticlesController create action to handle the submission of categories along with the article data. We do this through the article_params method we defined under **private**:

Here the debugger shows the categories getting passed through in **params**, by **id**.

The **id**'s are sent through in an array, and we can grab this array in the controller with the params method:

category_ids: ['1', '2', '3']

```
--- !ruby/hash-with-ivars:ActionController::Parameters
elements:
  utf8: "✓"
  authenticity_token: P7wefo/1vkqu69cqodYVTebwMDo2uvWHD
  article: !ruby/hash-with-ivars:ActionController::Para
    elements:
      title: ''
      description: ''
      category_ids: &1
      - '1'
      - '2'
      - '3'
      - ''
    ivars:
      :@permitted: false
      :@converted_arrays: !ruby/object:Set
  action: create
ivars:
  :@permitted: false
```

```
def article_params
  params.require(:article).permit(:title, :description, category_ids: [])
  #this is the secret method that stores the attributes we want to save an
    CREATE action.
```

articles_controller.rb

Save and test it out in the console. Next we will update views to incorporate categories.

The first think that comes to mind when considering how to display all the categories for a given article, is a simple block:

<% @article.categories.each do | x | %>

This works, but there is a shortcut to displaying ***all*** of a given instance:

<%= render @article.categories %>

Again, this will work as long as we want to print every instance. Of course, to use **render** we need a partial. Under views/categories create a new partial called _category.html.erb...

Add render and conditional to our show view:

```
<% if @article.categories.any? %>
  <p>Categories: <%= render @article.categories %></p>
<% end %>
```

The add the partial script; call the name of the instance and link

```
<span class="badge"><%= link_to category.name,
  category_path(category) %>  </span>
```

Do not be confused by the lack of explicit route for the **render** method. Rails docs has this to say about passing an instance variable to **render**:

> Every partial also has a local variable with the same name as the partial (minus the underscore). You can pass an object in to this local variable via the `:object` option:
>
> ```
> <%= render partial: "customer", object: @new_customer %>
> ```
>
> Within the `customer` partial, the `customer` variable will refer to `@new_customer` from the parent view.
>
> If you have an instance of a model to render into a partial, you can use a shorthand syntax:
>
> ```
> <%= render @customer %>
> ```
>
> Assuming that the `@customer` instance variable contains an instance of the `Customer` model, this will use `_customer.html.erb` to render it and will pass the local variable `customer` into the partial which will refer to the `@customer` instance variable in the parent view.
>
> **3.4.5 Rendering Collections**
>
> Partials are very useful in rendering collections. When you pass a collection to a partial via the `:collection` option, the partial will be inserted once for each member in the collection:

So in this case, @article.categories is referring to the categories found in an instance of article; And since categories is just the plural version of category, the render method seeks out a view and partial with the same name.

The method *to_partial_path* (on every model object that uses *ActiveRecord::Base*) gets called and looks at the Class name your model comes from, and returns a string along the lines of"#{classname.pluralize}/_#{classname.lowercase}"

Source: http://apidock.com/rails/v4.2.7/ActiveModel/Conversion/ClassMethods/_to_partial_path

If you dive into your code by running rails c you can grab any model you have and see where Rails expects to find the partial to represent the object.

```
rails c
Customer.first.to_partial_path
  #=> "customers/_customer"

Article.first.categories.first.to_partial_path
  #=> "categories/_category"
```

The article show page is now updated with category badges, so lets add them to the index view now. The index view is rendered from _article.html.erb:

```
<% if article.categories.any? %>
  <p>Categories: <%= render article.categories %></p>
  <!--render article.categories references viewa/categories/_category.html.erb-->
<% end %>
<% if logged_in? && (current_user == article.user || current_user.admin?) %>
```

Here the @articles object is passed to the partial where it is represented by **article**. Then **article.categories** is passed to another partial, categories/_category.html.erb.

The users/show.html.erb view is also rendered with articles/_article.html.erb:

```
<%= render 'articles/article', obj: @user_articles %>
```

Two for one; both views will display an article's categories.

Lets move on to the categories views. There is already a category index, but we also want the ability to click a category link from the index and see all the articles for that given category at categories/show.html.erb. For this we want to use the _article.html.erb partial as well, so start with the article/index template that renders this partial. There will be some changes though:

From:

```
<h3 align="center">Articles</h3>


<div align="center">
  <%= will_paginate %>
</div>
<!--Adds per page navigation-->

<%= render 'article', obj: @articles %>
<!--Within the article partial, the art

<div align="center">
  <%= will_paginate %>
</div>
```

articles/index.html.erb

To:

```
<h3 align="center"><%= "Category: " + @category.name %></h3>


<div align="center">
  <%= will_paginate @category_articles %>
</div>
<!--Adds per page navigation-->

<%= render 'articles/article', obj: @category_articles %>
<!--Within the article partial, the article variable will re

<div align="center">
  <%= will_paginate @category_articles %>
</div>
```

categories/show.html.erb

We use instance variables here that need to be defined in the CategoriesController **show** action:

```
def show
  @category = Category.find(params[:id])
  @category_articles = @category.articles.paginate(page: params[:page], per_page: 5)
end
```

The last little flourish, is just adding an article count to each category instance in in the index view:

```
<li>
  <small><p><%= pluralize(category.articles.count, "article") %></p></small>
</li>
```

Even though only an admin can edit/delete categories. We still want to make it easily accessible through UI. Lets add edit/update functionality to the CategoriesController. You know the drill:

```
def edit
  @category = Category.find(params[:id])
end

def update
  @category = Category.find(params[:id])
  if @category.update(category_params)
    flash[:success] = "Category successfully updated"
    redirect_to category_path(@category)
  else
    render :edit
  end
end
```

```
<h1 align="center">Update category</h1>

<%= render 'form' %>
```

As usual, the edit form is the same as the new form. Since the new form is already built, lets move it to a partial and use it for both.

```
<%= render "share/errors", {obj: @category} %>

<%= form_for(@category, :html => {class: "form-horizontal", role: "form"}) do |f| %>
  <div class="form-group">
    <div class="control-label col-sm-2">
      <%= f.label :name %>
    </div>
    <div class="col-sm-8">
      <%= f.text_field :name, class: "form-control", placeholder: "Enter category name",
        autofocus: true %>
    </div>
  </div>

  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <%= f.submit class: "btn btn-primary btn-lg" %>
    </div>
  </div>
<% end %>
<div class="col-xs-4 col-xs-offset-4">
  [<%= link_to "Cancel request and return to articles listing", articles_path %>]
</div>
```

The new **categories/_form.html.erb**.

```
<h1 align="center">Create new category</h1>

<%= render "form" %>
```

```
<h1 align="center">Update category</h1>

<%= render 'form' %>
```

Now to link the edit form to the show page and add restrictions:

```
<h3 align="center"><%= "Category: " + @category.name %></h3>


<div align="center">
  <% if logged_in? && current_user.admin? %>
    <div>
      <span class="badge cat-show-badge"><%= link_to "Edit Category Name",
        category_path(@category), class: 'category-name-badge' %></span>
    </div>
  <% end %>
  <%= will_paginate @category_articles %>
</div>
<!--Adds per page navigation-->

<%= render 'articles/article', obj: @category_articles %>
<!--Within the article partial, the article variable will refer to all the
  for a given category.-->

<div align="center">
  <%= will_paginate @category_articles %>
</div>
```