

Learn Ruby on Rails

Learn Web Development With Ruby on Rails

Who am I?

I am Harry and just like you I had never done any coding before. A complete beginner I wanted to learn how to create web apps such as a personal blog, Sign up forms, Authentication apps that allow users to login and logout. I wanted to know how to code useful apps that could be used for personal websites or even a business start up. So I started learning html and css and ruby and eventually ruby on rails. And now I want to share with you what I learned. I hope you learn something from these tutorials.

How to work through this book

If you are new to coding I recommend you read the book and type out all the code shown. This will take longer but it will stop you from glazing over. Trust me I know. Also if you type out the code examples and follow along then you will learn and retain the information far better.

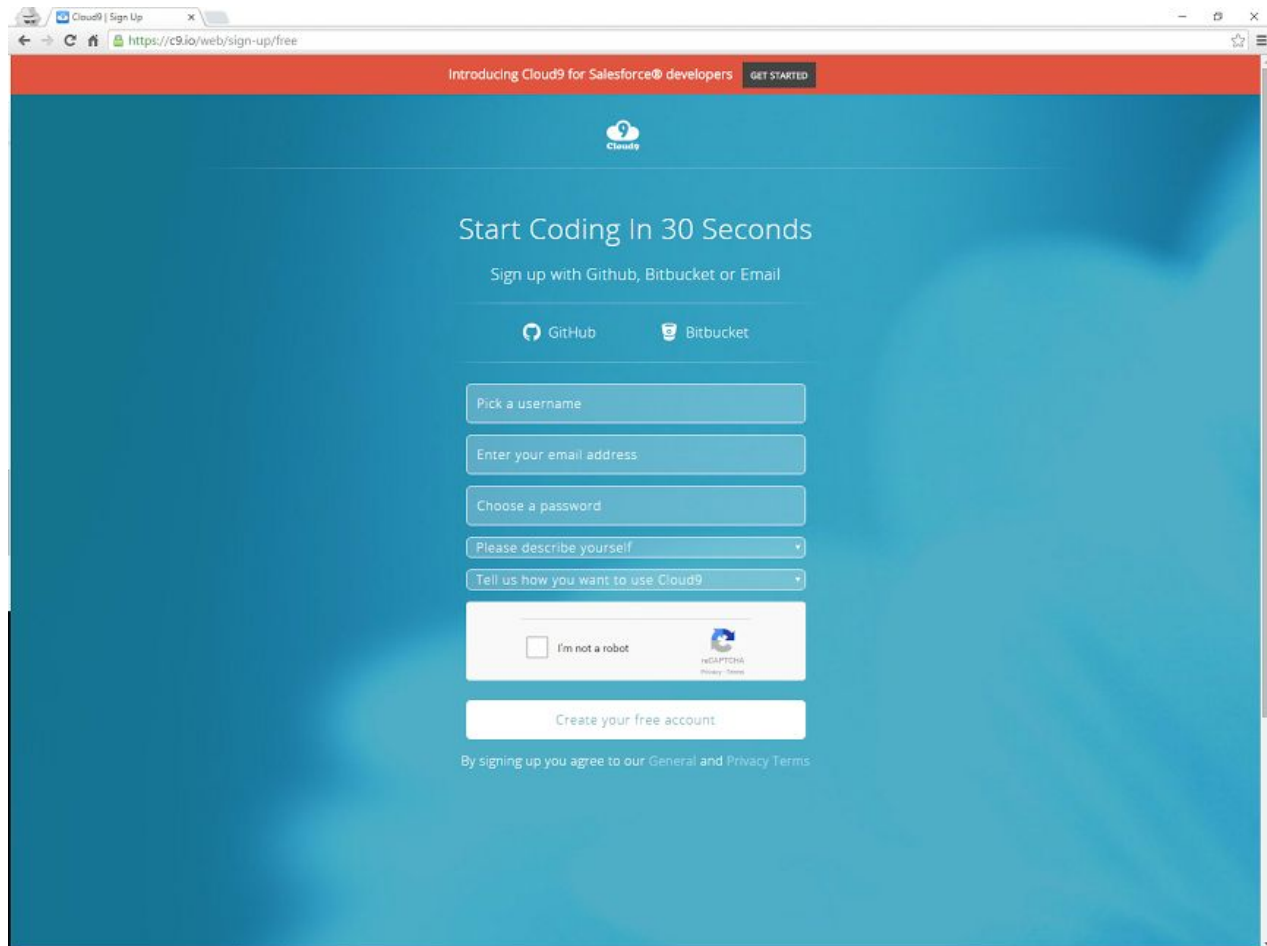
If you have coded before then feel free to jump to whatever you find most relevant.

Setting up our working environment

The working environment I am going to recommend is a cloud based environment, but why you might ask? Why not a local environment on your own computer? Well simply put everyone's computer will vary and so a cloud based environment is an easy default that all can use. If you want to set up a local environment then feel free, but this is a book on code tutorials and so I want you to be coding as quickly as possible. Also cloud9 provides a free as well as paid service and the free service is very good so it is what we will be using.

Go to <http://c9.io/> the website for cloud9 which is a cloud based coding environment.

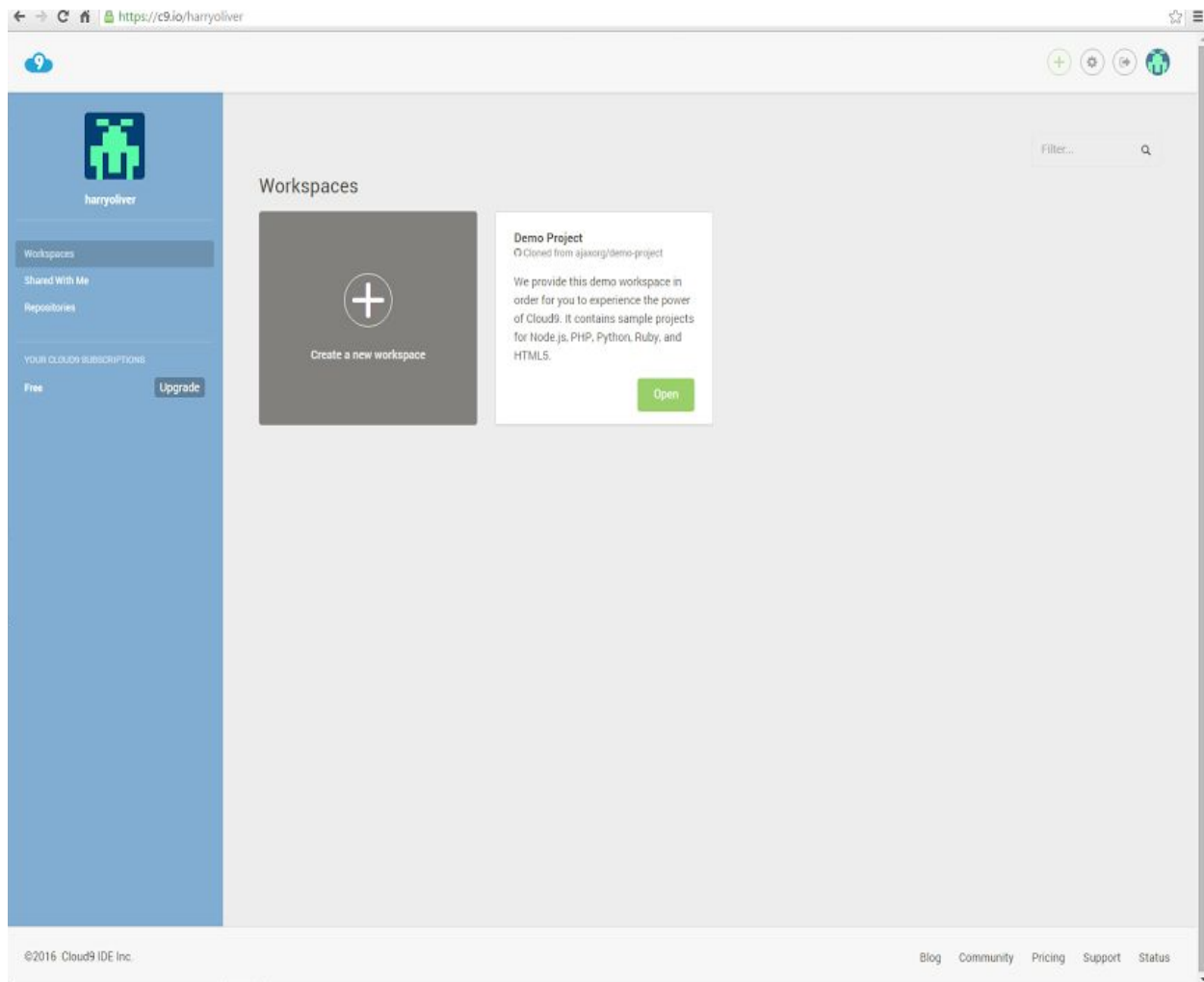
Click on the getting started button or sign up button.
You will see a sign up form, go ahead and sign up.



The screenshot shows a web browser window with the URL <https://c9.io/web/sign-up/free>. The page has a blue background and a red header. The header contains the text "Introducing Cloud9 for Salesforce® developers" and a "GET STARTED" button. The main content area features the Cloud9 logo, the heading "Start Coding In 30 Seconds", and the text "Sign up with Github, Bitbucket or Email". Below this are two buttons for "GitHub" and "Bitbucket". The sign-up form consists of several input fields: "Pick a username", "Enter your email address", "Choose a password", "Please describe yourself" (a dropdown menu), and "Tell us how you want to use Cloud9" (a dropdown menu). At the bottom of the form is a checkbox labeled "I'm not a robot" with a CAPTCHA logo. Below the form is a "Create your free account" button. At the very bottom, there is a link: "By signing up you agree to our General and Privacy Terms".

After you have put in your details and verified your email you will see a screen like the one below:

This screen is your home screen and holds all of your workspaces. Workspaces can be thought of as project folders. You can create many rails applications inside these workspaces, so don't think you have to create a new workspace for every rails application.



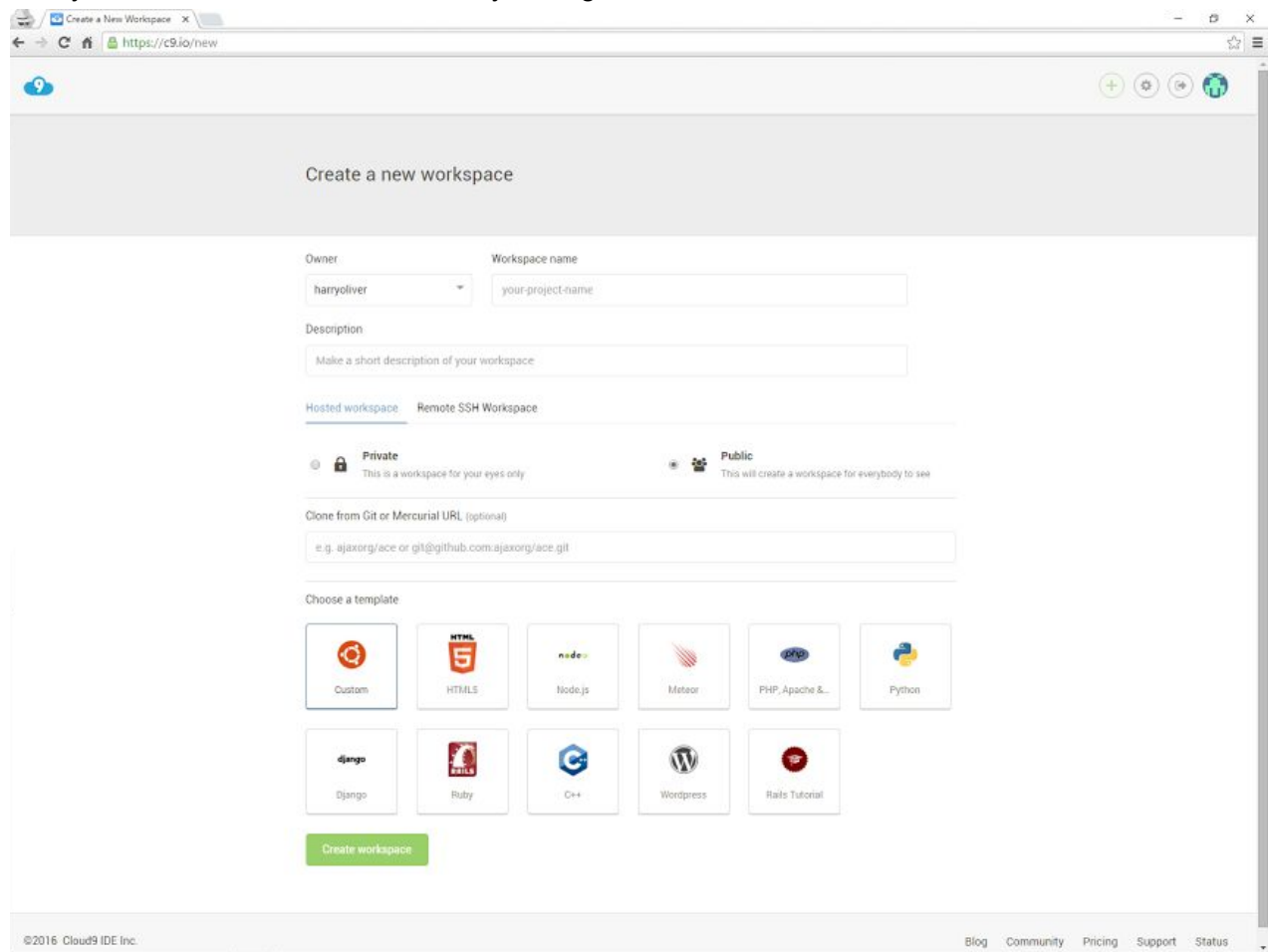
Click the Create a new workspace button

This will lead you to a screen that looks like this:

It provides you with various defaults to help set up your workspace however we will go for the custom template option.

Fill in the workspace name at the top, call it whatever you like and add a description. I called my workspace rails-tutorials. Leave the hosted workspace option as public. Again make sure you have selected the custom template option. It is the default option anyway. Finally click the create workspace button.

It may take 30 seconds but don't worry all is good ...

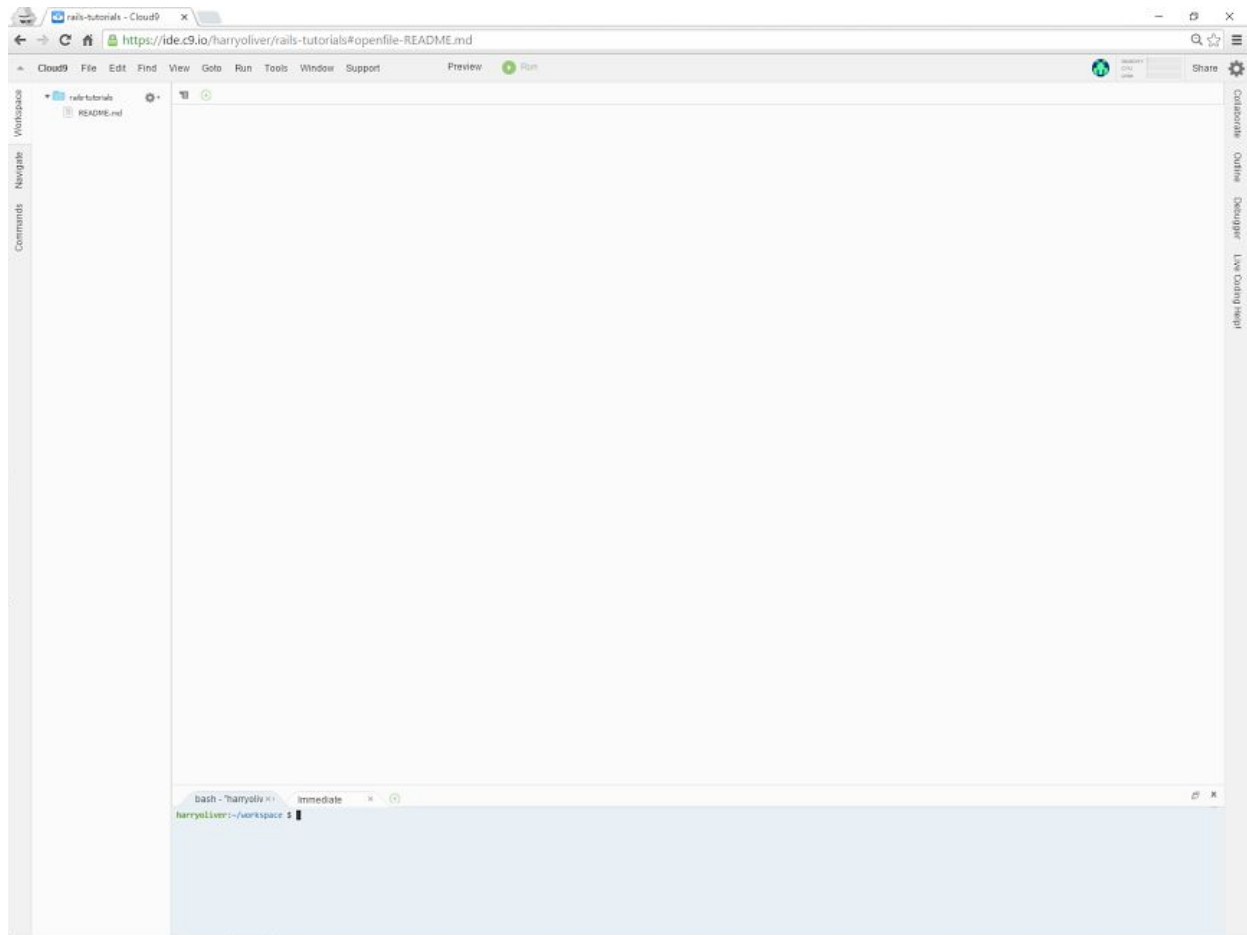


Okay once the workspace is created you will be greeted with your environment or IDE integrated development environment. Basically it is your own computer in the cloud and has everything pre installed that we need to get started.

You will see a toolbar at the top, file edit find etc. On the left hand side is a sidebar of your project folders. Since I named my workspace rails-tutorials I am in that workspace. The sidebar shows all the files and folders in that workspace. The only file in there at the moment will be a README.md file. As you can see it is also being displayed in the center of the page. In the center of the page near the top are some tabs. Close the README file by clicking the little x on the tab.

You will see the welcome tab is also open. You can adjust the settings as you wish, I recommend keeping it the same. The only change needed is where it says soft tabs 4, **Change the soft tabs to 2**. That is how far your code is indented when pressing the tab key on your computer. Now click close on the welcome tab.

At the bottom of your screen you will see your console, it will display your username followed by the folder/directory you are in, in this case the workspace directory.



Cool our development environment is now set up.

Lets check that the programming language ruby is installed and the Rails framework is also installed.

At the bottom in your console type the command:

```
ruby -v
```

Here is my output. Notice that the dollar sign is the prompt. You don't type it, it shows the end of your prompt or the beginning of where you type your commands. See how it shows I am in the workspace directory.

```
harryoliver:~/workspace $ ruby -v  
ruby 2.3.0p0 (2015-12-25 revision 53290) [x86_64-linux]
```


As you can see the output says ruby 2.3.0 which is fine. If you have a version that is newer then that will also be fine.

Next in the console type the command:

```
rails -v
```

You can see the output is Rails 4.2.5 again if your version is later than this version then it should be fine.

```
harryoliver:~/workspace $ rails -v  
Rails 4.2.5
```

Excellent the the ruby programming language we require is installed and so is the rails framework.

The command line

If you have experience with the command line already then feel free to skip this chapter. It is for the people who have rarely used or never used the command line at all. I will cover some basic commands and what they do, these commands will crop up throughout the tutorials so don't worry about memorizing them as you will see them enough throughout this book.

The Touch Command

At the bottom of your screen in the command line / console (I will use the names interchangeably) type:

```
$ touch hello.txt
```

Remember you don't type the dollar sign as that is the prompt.

If you look to the left of your screen where all your files and folders are you should see a file called hello.txt. If you don't then look for a small gear icon near the top of the files and folders section. Click it and you will see a drop down menu with an option to **refresh the file tree** which will show any new files or folders that have been created.

How does the touch command work?

The touch command creates a new empty file. It works by typing the command name which is touch followed by any options and then the file name.

```
touch [option] file_name
```

Don't worry this actually makes the command look more confusing. We won't be using any options with the touch command and so we can just type touch followed by the filename.

Type this touch command into the console:

```
$ touch file1 file2 file3
```

Now look at your files to the left, in one command you have touched 3 files into existence, pretty cool right?

The LS Command

Now type this command into your console:

```
$ ls
README.md file1 file2 file3 hello.txt
```

You should get a list of all the files you have created including the README file. The ls command is short for list and lists the files in the current directory. Pretty cool, we can also pass options or arguments to the ls command as well. We could pass the ls command the -a like this:

```
$ ls -a
./ ../ .c9/ README.md file1 file2 file3 hello.txt
```

When you add an option such as the -a which stands for all, you are saying list me the files but I want all of them. Even the hidden files. Hidden files have a single dot in front of them. .c9/ is a hidden directory, I know it is a directory / folder and not a file because of the forward slash at the end. That -a option that you used is known as a flag, that is the correct terminology for the options you pass to the commands. So **ls** is the command and the flag is **-a**, it is an option you pass to the ls command.

The PWD Command

This command prints your current location to the screen. You are in the workspace directory/folder so that is what will be printed to the screen, try this:

```
$ pwd
/home/ubuntu/workspace
```

The command stands for **Print Working Directory** and as you can see your current directory is the workspace directory that cloud9 has given you. You can also see that the workspace directory is in a directory called ubuntu which is in another directory called home. By default cloud9 actually show what directory you are in, you can see this by looking at your prompt.

```
~/workspace $
```

The prompt shows the workspace directory.

The Mkdir Command

This command creates a new directory or folder. Remember how you used the touch command to create a new file? This command is used to create a new folder.

Try typing this into your console:

```
$ mkdir myfolder
```

The mkdir command is short for **make directory** and is quite self explanatory. If you look to the left at your files and folders you will see a new folder called myfolder. It is empty at the moment so we will add some files to it. We already know how to create a new file with the **touch** command but how do we get inside that new folder we just created?

The CD command

The cd command is used to change the directory you are in. It stands for **change directory**. If you look at your prompt you will see you are in the workspace directory. Try typing this to change to the myfolder directory.

```
$ cd myfolder  
~/workspace/myfolder $
```

See how the prompt updates and you are now in the myfolder directory. So how do I get out? The way to get back to your workspace directory is to type:

```
$ cd ../
```

That is the cd command then a space then two dots and a forward slash. This jumps you out of one folder and into the one above. Since we were in the workspace directory when we created the myfolder directory the workspace directory is the one above.

If you type cd on its own then you will go to the root directory. In this case it is represented by a tilde ~ on cloud9. If you list all the files and folders with the ls command you will see the workspace directory where you have been working.

```
~/workspace $ cd  
~ $ ls  
lib/ workspace/  
~ $ cd workspace  
~/workspace $
```

I wanted to show new users of the command line how to use the commands cd and ls here so that you don't get stuck. Remember cd ../ is used to jump out of the current folder you are in.

The RM command

The rm command is short for remove and it deletes files and folders. Try typing:

```
$ rm file1
```

You will see that the file named file1 has now been deleted.
Now try to delete the folder named myfolder.

```
$ rm myfolder
```

You will probably get a message like this:

```
$ rm myfolder  
rm: cannot remove 'myfolder': Is a directory
```

Okay you cannot delete myfolder as it is a directory.

The next command will remove the folder, try the command again but add the -r flag.

```
$ rm -r myfolder
```

As you will see the folder is now deleted. The -r is short for recursively, It basically means that any files inside that directory will also be deleted and are gone forever. That is why we had that original error message. We were being protected from accidentally doing something stupid. We didn't actually have any files in our myfolder directory but you get the idea.

Okay to get back to square one type:

```
$ rm file2 file3 hello.txt
```

This removes all the files we created, we have just named them one after another instead of typing the rm command multiple times.

During the rest of the tutorials I will mention what commands need to be used so you don't need to memorize these commands, They are just there to show you the basics of navigating, creating and deleting files.

If you are brand new to any of this try experimenting and creating your own files and folders. The best way to learn is to get stuck and then get unstuck. Hopefully not taking too long in the process.

Your First Rails App

A Static Webpage

Okay time for you to create your first rails app. It will be very simple and only display a page, however hopefully you will learn a bit about the structure of a rails app (the way all the files and folders are laid out) as well as a few of the basic rails commands (That we will use to generate things for us, similar to the commands we have been using above) and a little bit about rails conventions (rails does things in a certain way to save you time).

In your workspace directory type the rails command to generate a new rails app:

```
Creating your first rails app
/workspace

$ rails _4.2.5_ new static_page
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  create  app/helpers/application_helper.rb
  create  app/views/layouts/application.html.erb
  create  app/assets/images/.keep
  create  app/mailers/.keep
  ...
```

You will get a nice big list of things that have been created and installed, much longer than the list here, I have shortened it to save space.

Lets look at that rails command used to generate a new application:

```
$ rails _4.2.5_ new static_page
```

We start with the word rails as we are using a rails command, then we have an argument we passed specifying a version number. Let's get rid of that for a minute and take another look at the command.

```
$ rails new static_page
```

Okay cool this is starting to look a little bit more like english I can understand! See the words static_page, that is the name of the directory or rails app. It is what we have called this project.

```
$ rails new
```

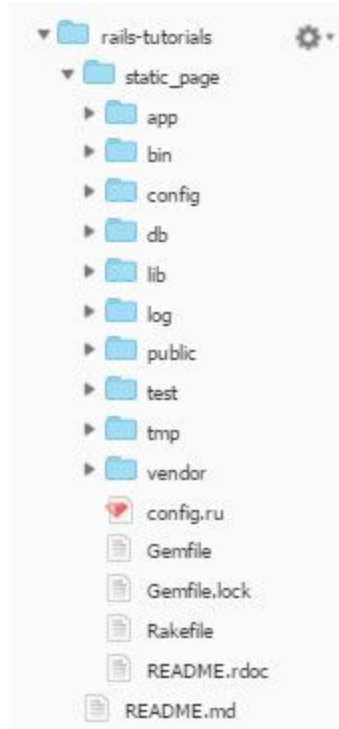
So rails new is the actual command. We use this command to generate a new application. A new what though? That is why we named the application static_page. It could have been named anything but it made sense to call it something relevant. So back to the original command, why the underscores and numbers?

```
$ rails _4.2.5_ new static_page
```

The `_4.2.5_` we passed to the rails new command is a version number. It means use this version of rails. The reason I have done this is so that you can follow along with as few hiccups as possible. If you were making your own app you might just do **rails new myapp** without a version number.

Okay cd into your new rails app, and on the left hand side where the files and folders are displayed click your new folder (the little triangle) to open up the new application.

```
$ cd static_page  
~/workspace/static_page $
```



Do you see the app/ directory, config/ directory these two folders are the ones we will use most for this tutorial. Okay in order for us to check that everything has installed correctly let's start up the server and run the rails application. Type the command:

```
Creating your first rails app  
/workspace/static_page
```

```
$ rails server -b $IP -p $PORT
```

The actual command is rails server all on its own, however we have a -b flag and a global variable \$IP followed by a -p flag and a \$PORT global variable. When using cloud9 you add those flags and global variables to run the server. When using cloud9 if you type just rails server it won't work. The extra flags and global variables are a little bit of a pain to type, so feel free to copy this command into your console.

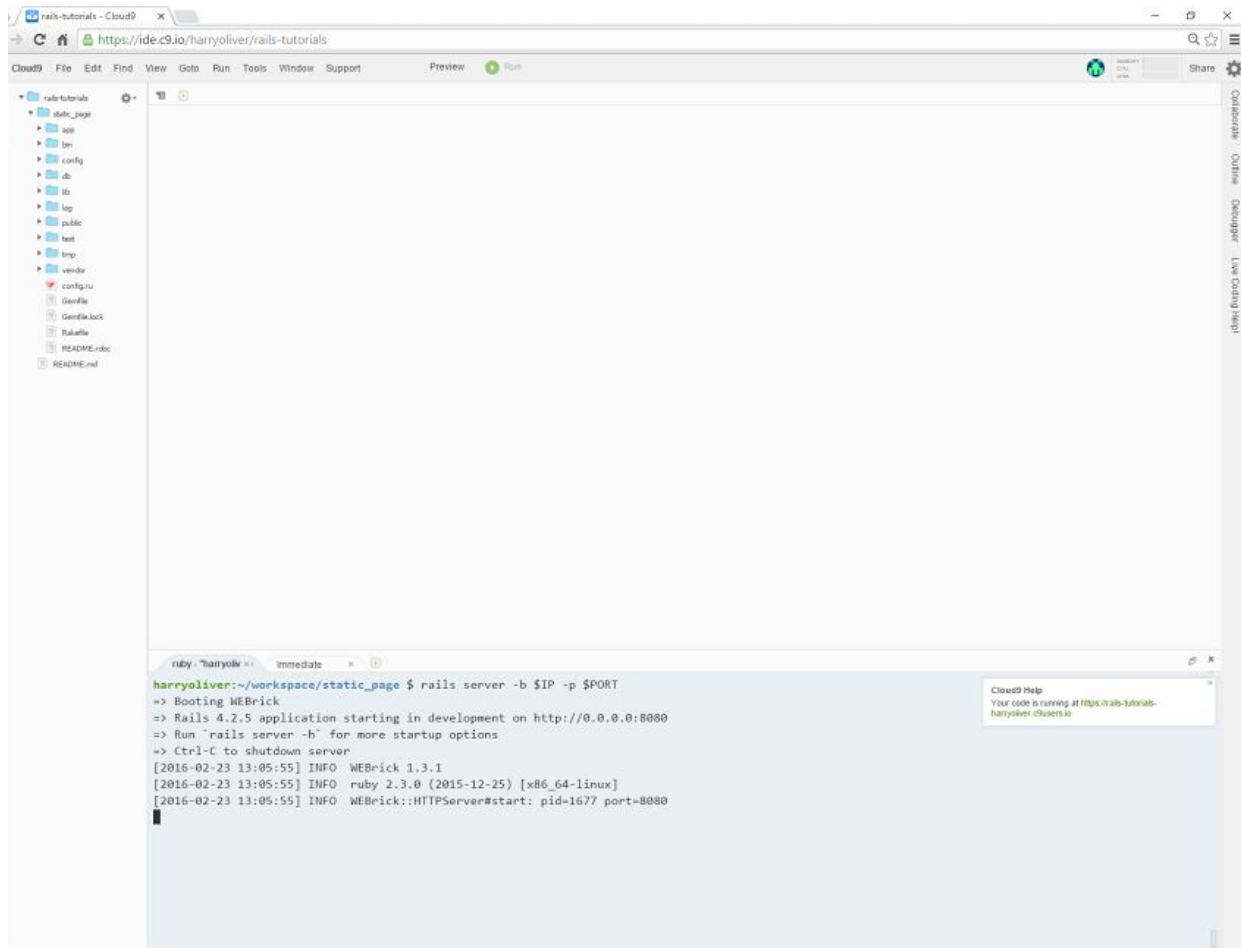
```
Creating your first rails app  
/workspace/static_page
```

```
$ rails server -b $IP -p $PORT  
=> Booting WEBrick  
=> Rails 4.2.5 application starting in development on http://0.0.0.0:8080  
=> Run `rails server -h` for more startup options  
=> Ctrl-C to shutdown server  
[2016-02-23 13:05:55] INFO WEBrick 1.3.1  
[2016-02-23 13:05:55] INFO ruby 2.3.0 (2015-12-25) [x86_64-linux]
```

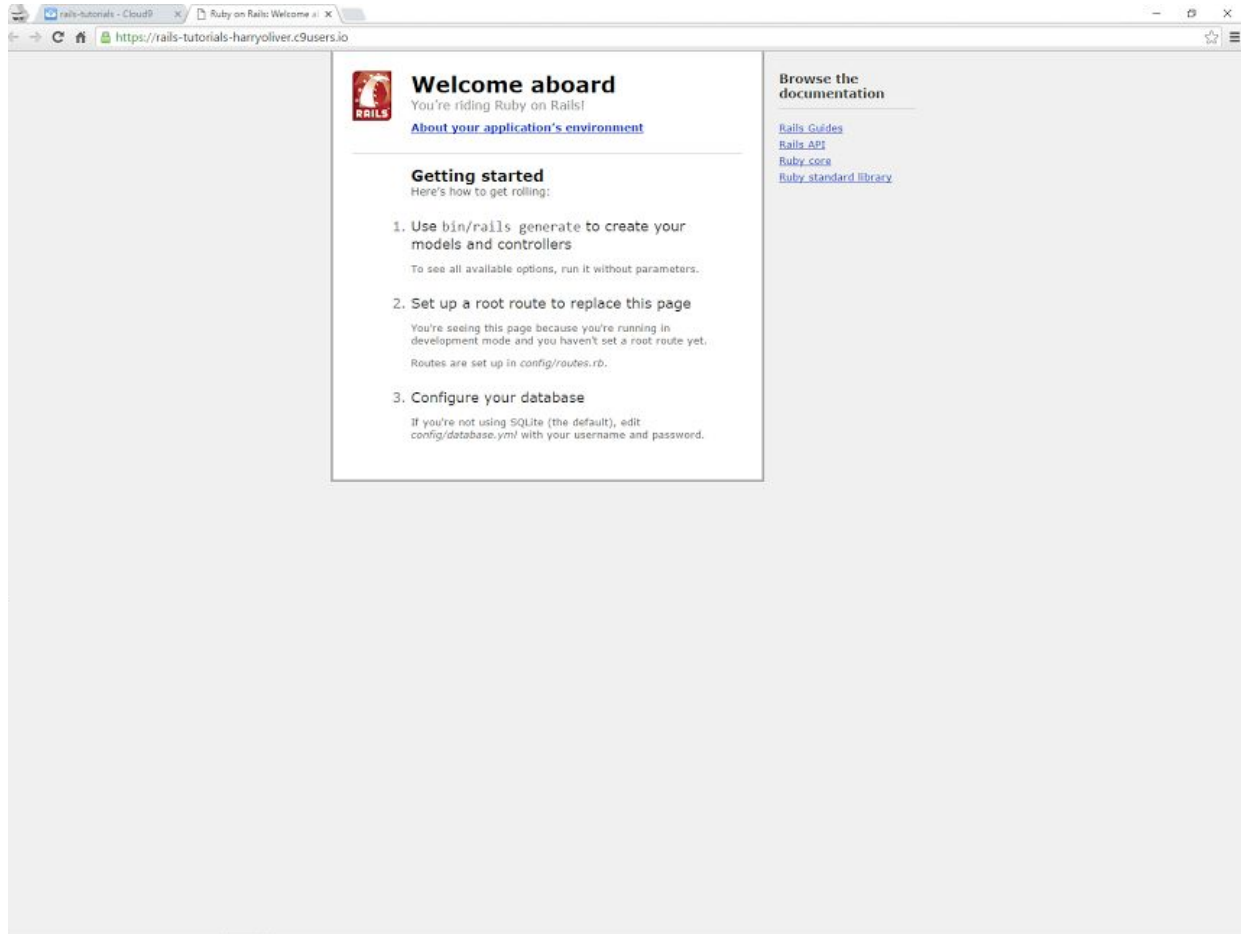


```
[2016-02-23 13:05:55] INFO WEBrick::HTTPServer#start: pid=1677 port=8080
```

Look at the output it says booting webrick, this is the default rails server. Then a bit further down it says CTRL-C to shutdown the server. That is the CTRL key plus the letter C key on your computer.



See the cloud9 help popup on the right there, click the link and it will open your rails app in a new tab, Or in the console output after the booting webrick line, click the <http://0.0.0.0:8080> line where it says application starting in development. You should see this:



This is the default rails application, Feel free to check out the link on the right such as rails guides, rails api etc. Click CTRL-C to exit the server (you need to click in the console first) and close the extra tab. Cool you have created a default rails application and run the server and viewed the application. LET'S GET CUSTOMIZING!

Generating the Controller

We start by generating the controller, as it's name suggests it controls things. If we type in a url it will control what we see (Don't worry a greater explanation will follow). Type the command in yellow, as you can see the name of the rails controller is Pages.

```
Creating your first rails app
/workspace/static_page
```

```
$ rails generate controller Pages
Running via Spring preloader in process 1711
  create  app/controllers/pages_controller.rb
  invoke  erb
  create  app/views/pages
```

```
invoke test_unit
create test/controllers/pages_controller_test.rb
invoke helper
create app/helpers/pages_helper.rb
invoke test_unit
invoke assets
invoke coffee
create app/assets/javascripts/pages.coffee
invoke scss
create app/assets/stylesheets/pages.scss
```

```
$ rails generate controller
```

The actual command is the one above, we tell rails generate a controller. We pass the command the name of the controller. In this case Pages. Look at what gets created, a **pages_controller.rb** file that is located in the directory **app/controllers/pages_controller.rb**. Also look at the app/views/pages directory. A pages directory has been created. In your navigator on the left with all your files and folders try finding these files and directories mentioned.

We used the command rails generate controller Pages with a capital p. The file that got generated though was pages_controller.rb. Rails has automatically converted the file name to snake case (The use of underscores).

Start up your server again:

```
Creating your first rails app
/workspace/static_page
```

```
$ rails server -b $IP -p $PORT
```

In the url at the top of the page add **/pages** to the end of the url. You will see a routing error. We will solve this shortly. Basically we need to tell rails all the routes/urls that we will use.

Open up the file pages_controller.rb that you have just generated. See how the pages_controller.rb is nicely located in the app folder then the controllers folder and then you see the file we want pages_controller.rb.

```
Creating your first rails app
/workspace/static_page/app/controllers/pages_controller.rb
```

```
class PagesController < ApplicationController
end
```

Let's edit this file a little bit. See we have a class of PagesController that got generated, the less than sign < means this class inherits from the ApplicationController which is part of rails. Also see that the class is defined with the class word followed by the name of the class and then on a new line the end word. Between this is where we add our code.

```
Creating your first rails app  
/workspace/static_page/app/controllers/pages_controller.rb
```

```
class PagesController < ApplicationController  
  
  def home  
  end  
  
end
```

Add the home action. The thing I found difficult about learning rails was that things had different names, for example the home **action** is actually a ruby (programming language) method. Rails is built on top of ruby and the way you define a method in ruby is with the def and end keywords. In other programming languages a more common name for method is a function. So technically you could call what you just wrote a home method or function. However we will stick with the correct terminology here and call it a home action. Also you will see that the method itself is empty. This is okay, in normal ruby programming this would do nothing, however with rails since this PagesController class inherits from the ApplicationController from rails instead of doing nothing by default it renders a view. We will create the view in a minute. **First Save The File By Pressing CTRL-S a small circle in the tab near the top will become an x. Always save every change you make!**

Look at the pages_controller.rb file, it is named with snake case or underscores but the class inside the file PagesController has been named in camelcase where every first letter of a word is capitalized. This is another way rails does things and it was automatically done when we generated the PagesController. Don't type that bit below.

```
$ rails generate controller Pages
```

*Since the command states we want to generate a **controller** we don't need to type the word controller after pages, it is converted for us. Also if you look at the word Pages note I used camelcase for the name of the controller. So if I had called the controller website pages it would have been named: WebsitePages.*

Adding the Route

Now that we have our controller with the home action, let's add the route. If you remember when we typed in /users on to the end of the url we got an error, a routing error. Rails didn't know where to look. So let's fix the problem. Go to your routes.rb file located in the config folder.

```
Creating your first rails app
/workspace/static_page/config/routes.rb
```

```
Rails.application.routes.draw do
  # The priority is based upon order of creation: first created -> highest priority.
  # See how all your routes lay out with "rake routes".

  # You can have the root of your site routed with "root"
  # root 'welcome#index'

  # Example of regular route:
  # get 'products/:id' => 'catalog#view'

  # Example of named route that can be invoked with purchase_url(id: product.id)
  # get 'products/:id/purchase' => 'catalog#purchase', as: :purchase

  # Example resource route (maps HTTP verbs to controller actions automatically):
  # resources :products

  # Example resource route with options:
  # resources :products do
  #   member do
  #     get 'short'
  #     post 'toggle'
  #   end
  #
  #   collection do
  #     get 'sold'
  #   end
  # end
  ...
end
```

Okay there is a lot going on but they are all comments apart from the top and bottom lines. Add this code to the file:

```
Creating your first rails app
/workspace/static_page/config/routes.rb
```

```
Rails.application.routes.draw do
  root 'pages#home'
  # The priority is based upon order of creation: first created -> highest priority.
  # See how all your routes lay out with "rake routes".

  # You can have the root of your site routed with "root"
  # root 'welcome#index'

  # Example of regular route:
  # get 'products/:id' => 'catalog#view'

  # Example of named route that can be invoked with purchase_url(id: product.id)
  # get 'products/:id/purchase' => 'catalog#purchase', as: :purchase

  ...
end
```

I have not deleted anything, only added some code to line 2. The root word tells rails what page to load first when you go to the application. A home page is what you would usually go to, this can of course be any page you create but in this tutorial we only have one page so don't worry.

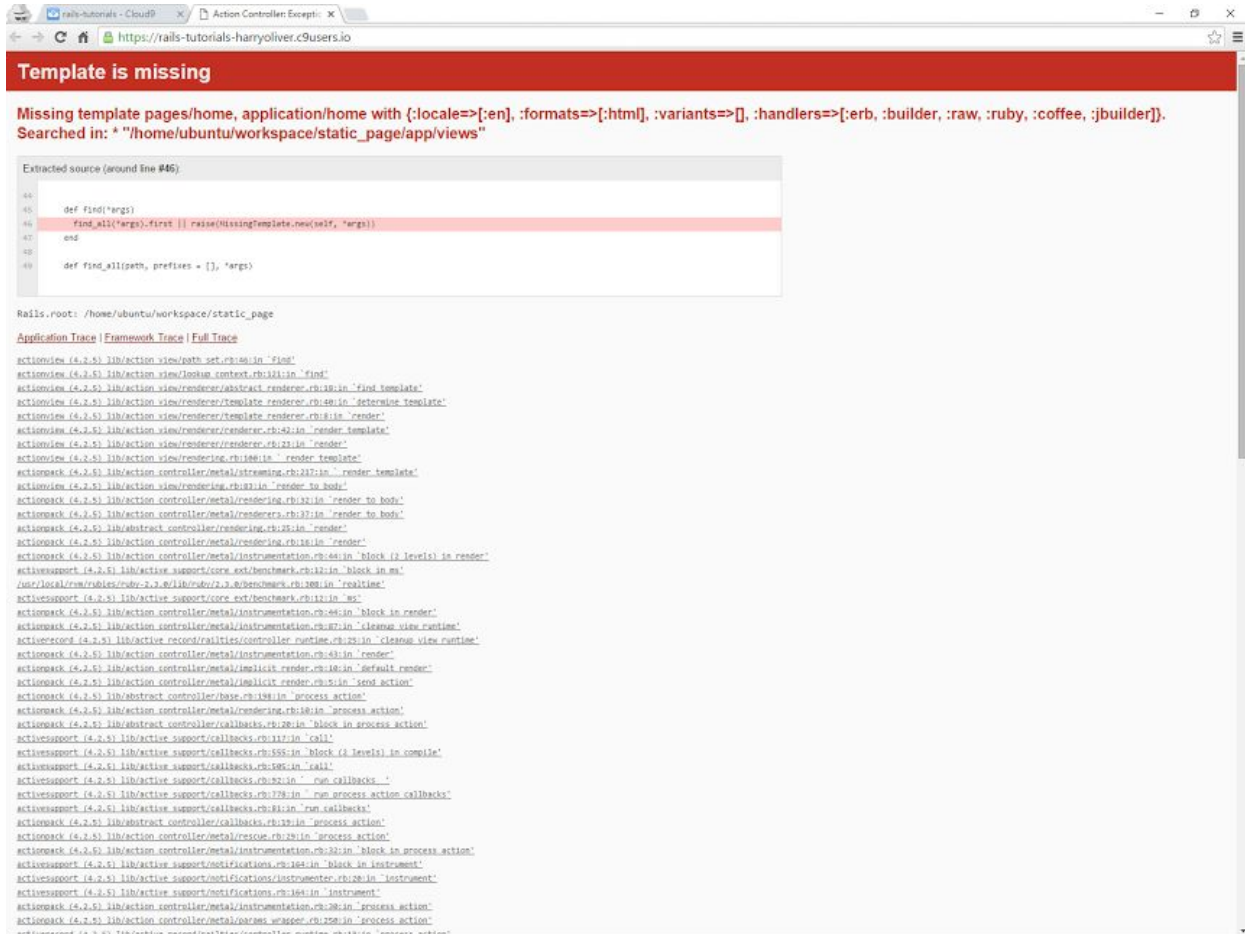
Look at the line 'pages#home', remember we generated the pages controller we are telling rails use that same pages controller. Open up that pages_controller.rb file and you will see the home action we created. After the word pages we add a hash or pound sign (rails syntax) and then we say use this home method I wrote.

So hopefully that routing line makes a little bit more sense. Root me to the pages_controller and I want to see the home action. So it is the controller first then a pound sign and then the action name.

Start up your server:

```
Creating your first rails app
/workspace/static_page
$ rails server -b $IP -p $PORT
```

Okay we having a missing template error. Not a problem. Remember I said even though our home action is empty rails still does stuff by default. Well by default it renders a view. Time to create that view.



Press CTRL-C to close the server.

Creating The View

Look in the app folder then views folder and you will see an empty pages folder. This folder was generated when we used the rails generate controller Pages command. It automatically generated a folder to put all of our views related to the pages_controller.

You have two ways to create the view file that we need. You can use the touch command covered earlier or you can right click and hit create new file. Either way we need a file named **home.html.erb** The file is named home because we created a home action remember? The HTML is to let us know it is a html file and the .erb lets us know we can use embedded ruby a type of syntax that lets us type ruby code in a html file as well as normal html tags such as `<p></p>` and `<div></div>`.

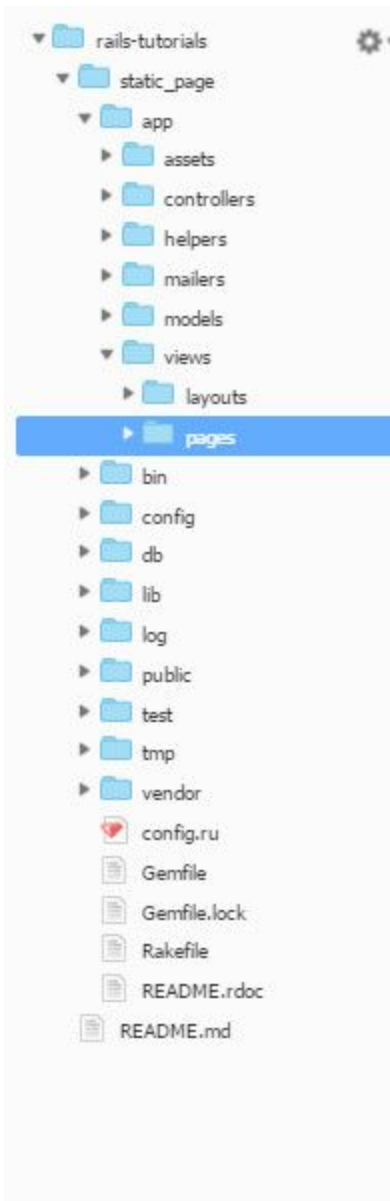
Method 1

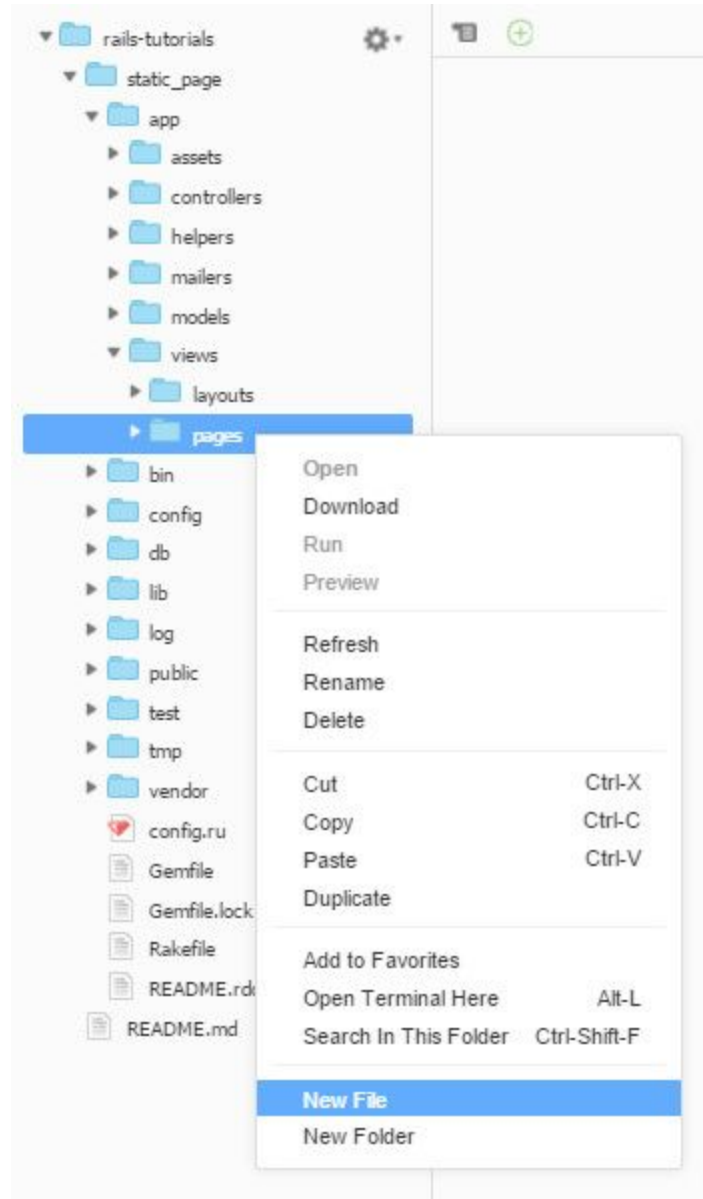
```
~/workspace/static_page $ touch app/views/pages/home.html.erb
```

See how we specify the path where we want to create the file, We want it in the app folder then in the views folder then our empty pages folder and then the actual file name itself **home.html.erb** Remember rails is built with ruby, the file name therefore has the .erb extension so that we can use embedded ruby in the file if we want to.

Method2

Or you can right click and name the new file home.html.erb whichever is easier for you:





Okay open up your newly created home.html.erb file and add some html to the page.

```
Creating your first rails app  
/workspace/static_page/app/views/pages/home.html.erb
```

```
<h2>Welcome Home</h2>  
<p>After your long journey through rocky mountains and dry dust bowls you have finally  
neared the end.</p>
```

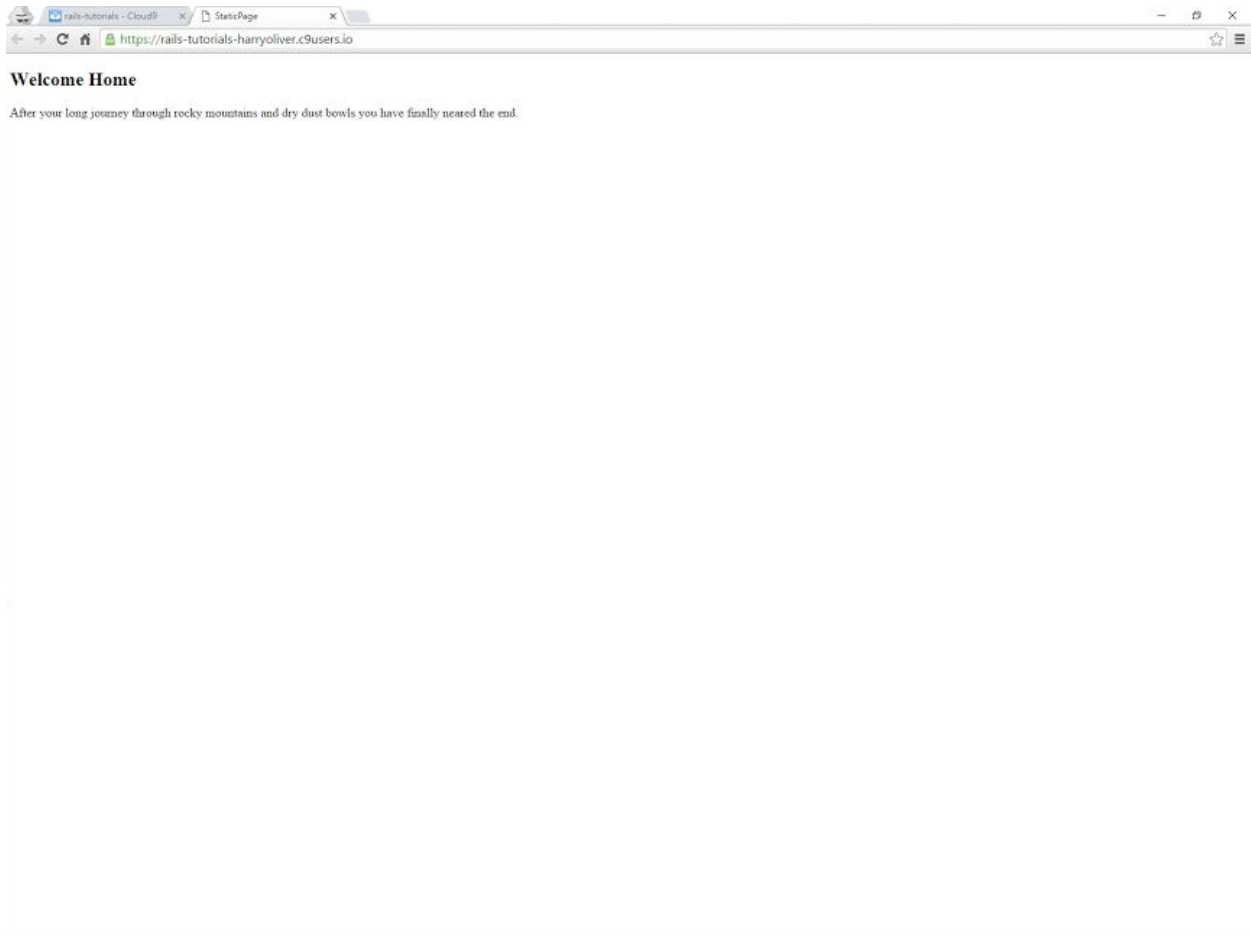
Feel free to copy and paste the html. Once again don't forget to save by pressing CTRL-S You can do file then save but that takes too long. I have not mentioned saving very much but make sure you save in order to see the changes you have made.

Okay start up your server again:

```
Creating your first rails app  
/workspace/static_page
```

```
$ rails server -b $IP -p $PORT
```

You should have something similar to this:



Well done you have created your first rails application. So what exactly have you done then? You started by generating a Pages controller and adding a home action to the PagesController class. You then defined the root route which meant your application would by default load the home action of the Pages controller, And finally you added the home.html.erb view. You have created a static rails app. I know it doesn't seem very impressive but these are all things you will do multiple times when creating rails apps so it helps to nail the basics!

If you are still in your rails application folder you can cd out of it back into your workspace. Onto application number 2.

```
~/workspace/static_page $ cd ../  
~/workspace $ pwd  
/home/ubuntu/workspace
```

WELL DONE!



A Styled App

The second app you build will be similar to the first app. It will help drum home various rails commands and things you need to remember. The start of this app is going to have a little less explanation compared to the first app. This is so that we can speedily get set up and continue learning. Then when we are at a new point I will start the explanations again. This first bit won't be anything too new, it will be very similar to the previous chapter.

You may be wondering why not just continue with the first app and build on top of that? Well in my opinion a better way to learn is to build multiple apps because you are far more likely to be typing the commands hundreds of times compared to just a few times. For example you don't actually type the rails new command that often, only at the start of an application. However it is still important to remember the command otherwise you won't be creating any new apps hahahaha.

Make sure you are in the workspace directory:

```
~/workspace $ pwd
/home/ubuntu/workspace
```

Generate a new rails app called styled_app

```
A styled app
/workspace

$ rails _4.2.5_ new styled_app
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  create  app/helpers/application_helper.rb
  create  app/views/layouts/application.html.erb
  create  app/assets/images/.keep
  create  app/mailers/.keep
  ...
```

Change directory into the new app:

```
$ cd styled_app/
~/workspace/styled_app $
```

Run the server: Check all is fine.

```
A styled app
/workspace/styled_app

$ rails server -b $IP -p $PORT
```

Generate a pages controller:

```
A styled app
/workspace/styled_app

$ rails generate controller Pages home
Running via Spring preloader in process 3734
  create  app/controllers/pages_controller.rb
  route  get 'pages/home'
  invoke  erb
  create  app/views/pages
  create  app/views/pages/home.html.erb
  invoke  test_unit
  create  test/controllers/pages_controller_test.rb
  invoke  helper
  create  app/helpers/pages_helper.rb
  invoke  test_unit
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/pages.coffee
  invoke  scss
  create  app/assets/stylesheets/pages.scss
```

Okay time for a closer look at that generate command. Most of the command should make sense, you are telling rails to generate a controller named Pages, however if you look we have added the word home. We have used the same command but added an argument.

```
rails generate controller NAME [action]
```

The argument we added was an action. Remember inside the controller we can add ruby methods however they are called actions. Remember the home action from the previous chapter. This is a quicker way of generating the same thing.

Look at what got created, a route 'pages/home' as well as a home.html.erb template in its appropriate pages folder which is again inside the views folder. At the very top the pages_controller.rb got created and if you look at the bottom you can see some app/assets/stylesheets got generated which we can use to style the application with css.

Look at your pages_controller.rb file:

The home action automatically got generated.

```
A styled app
/workspace/styled_app/app/controllers/pages_controller.rb
```

```
class PagesController < ApplicationController
  def home
    end
end
```

Look at your home.html.erb file it also got automatically generated:

```
A styled app
/workspace/styled_app/app/views/pages/home.html.erb
```

```
<h1>Pages#home</h1>
<p>Find me in app/views/pages/home.html.erb</p>
```

Now take a look at your routes.rb file:

```
A styled app
/workspace/styled_app/config/routes.rb
```

```
Rails.application.routes.draw do
  get 'pages/home'

  # The priority is based upon order of creation: first created -> highest priority.
  # See how all your routes lay out with "rake routes".

  ...
end
```

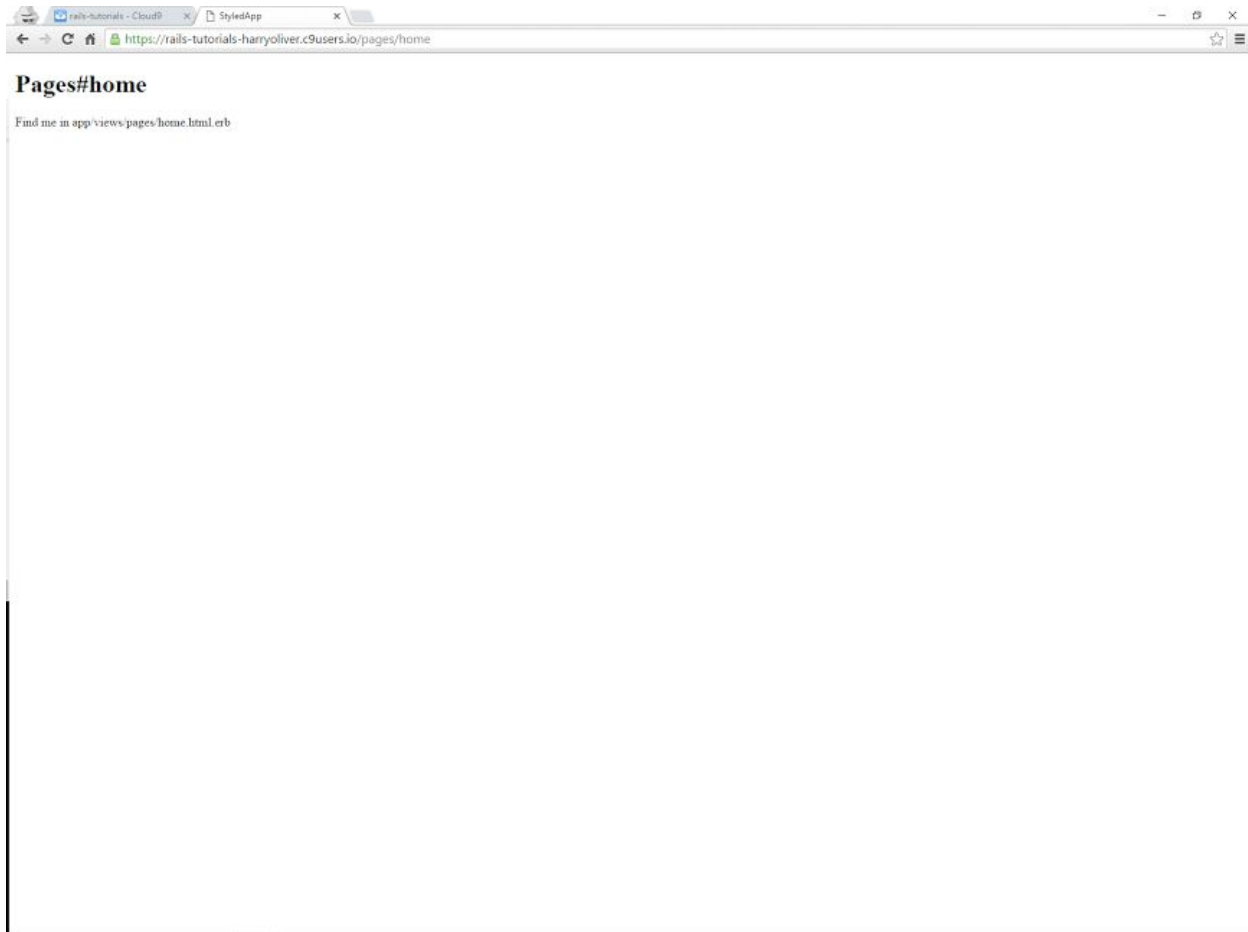
A get route also got added automatically. This is a bit different than the root route we used earlier. However it still uses the pages controller and home action in that controller.

Run the server:

```
A styled app
/workspace/styled_app
```

```
$ rails server -b $IP -p $PORT
```

Add **/pages/home** to the end of your url to view the about page, This comes from the routes.rb file where it creates a get request for the pages controller and the home action which defaults to rendering the home.html.erb view file. You should see the home page first:



Close the server CTRL-C
Remember to be saving your work.

Add an about action to the pages_controller.rb file:

```
A styled app  
/workspace/styled_app/app/controllers/pages_controller.rb
```

```
class PagesController < ApplicationController  
  def home  
    end  
  
  def about  
    end  
end
```

Add an about.html.erb file to your app/view/pages directory:

I use the touch command here but remember you can right click if you like.

```
A styled app
/workspace/styled_app
```

```
$ touch app/views/pages/about.html.erb
```

Update the routes.rb file:

```
A styled app
/workspace/styled_app/config/routes.rb
```

```
Rails.application.routes.draw do
  get 'pages/about'
  root 'pages#home'

  # The priority is based upon order of creation: first created -> highest priority.
  # See how all your routes lay out with "rake routes".

  ...
end
```

I remove the get route and replace it with two lines above. The root of the application becomes the pages controller home action which defaults to rendering the home.html.erb file. If you type **/pages/about** on to the end of your url you will get to the about.html.erb file, this file will be blank as we have not added anything to it yet. The pages/about route uses the keyword get. This is a type of http request between a web browser and server. A get request retrieves data from a particular source. In this case we want to get the about.html.erb file. To read more on get requests and post requests check out [w3schools](#).

Okay where have we got to with this app? We have created the same home page like we did in the previous app and we have also added an about page as well, Hopefully you did this in a much shorter time than the last chapter.

Adding Links

Add the following lines to your home.html.erb file:

```
A styled app
/workspace/styled_app/app/views/pages/home.html.erb
```

```
<h1>Pages#home</h1>
<p>Find me in app/views/pages/home.html.erb</p>
<%= link_to 'ABOUT' %>
```

Save your work and run the server:

```
A styled app
/workspace/styled_app
```



```
$ rails server -b $IP -p $PORT
```

Look at your home page and you will see we have added a link with the text ABOUT. However if you click it then it won't go anywhere. Why not? Okay look at the line you added, you used embedded ruby.

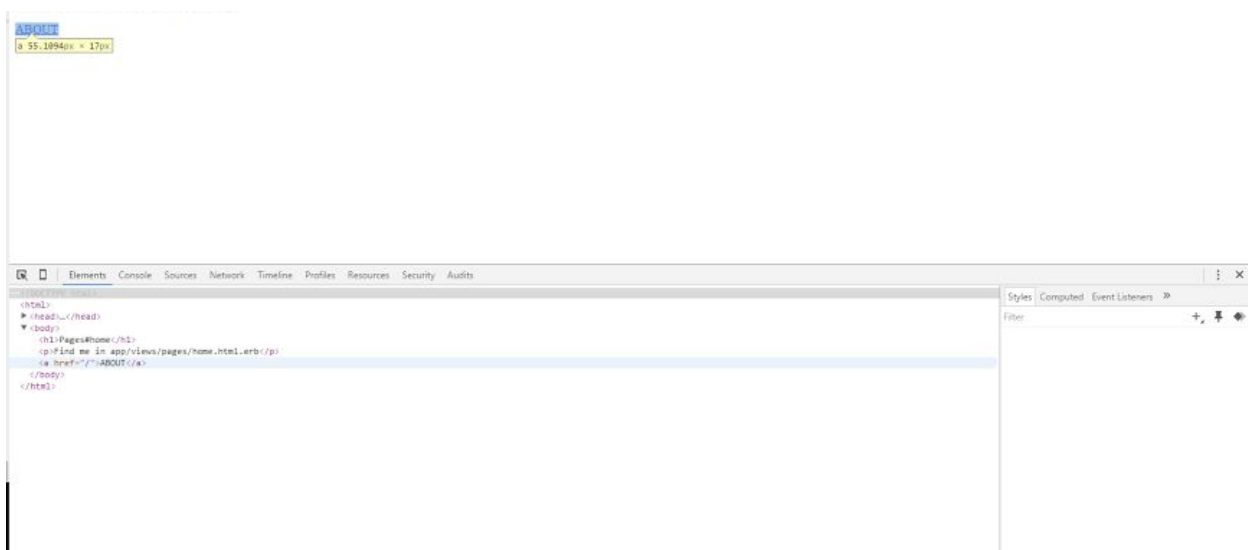
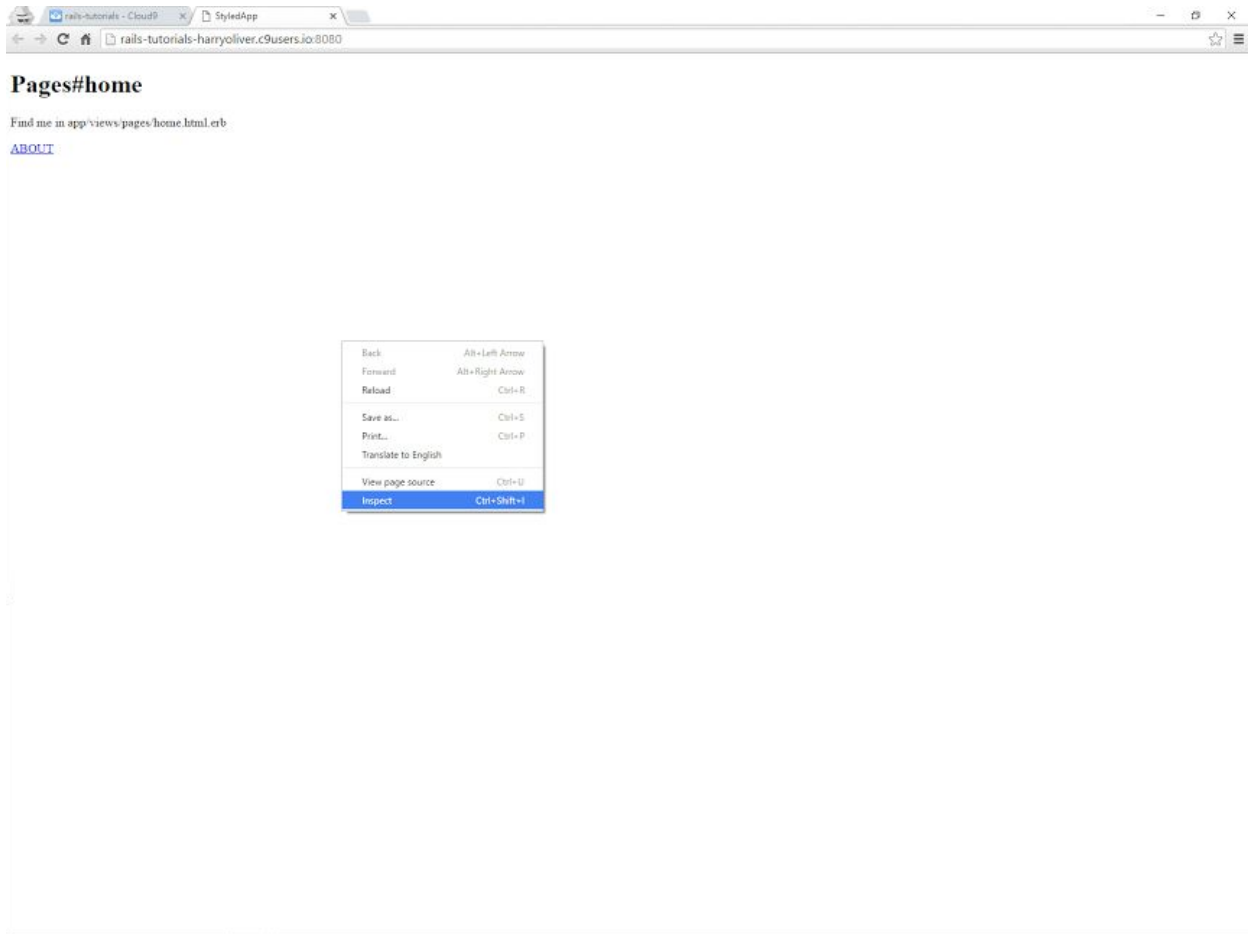
Embedded ruby uses either two formats

1. `<% %>`
2. `<%= %>`

Version one is a less than sign and a percentage sign. it is used for processing logic such as if statements. Version two is a less than sign a percentage sign and an equals sign. It is used for **OUTPUTTING** to the screen. If you want to see the result then you use the embedded ruby with the equals sign. As you build more apps this will become a lot clearer.

Inside the embedded ruby we use a rails helper method. It is called that because it is helping generate a link. The method is `link_to` it takes the name of the link in this case we named it ABOUT and it also takes the path to go to as an argument. Before we add that though let's inspect our home page.

Right click on the page and select inspect:



If you open up the body tags and look at the about anchor link you will see the link_ to just generates a normal link in html. If you look at the href='/' you will see we haven't provided a destination so the link has defaulted to forward slash which just returns us to the home page. Okay time to update the link to take us to the about page.

Update the home.html.erb file:

```
A styled app
/workspace/styled_app/app/views/pages/home.html.erb

<h1>Pages#home</h1>
<p>Find me in app/views/pages/home.html.erb</p>
<%= link_to 'ABOUT', pages_about_path %>
```

You add a comma followed by the text `pages_about_path`, This line will take you to the about page. If you restart the server and inspect the anchor element again you will see the hyperlink reference now points to the pages controller and about action: `href='/pages/about'`.

Where does the `pages_about_path` come from?

You defined some routes in your `routes.rb` file so the line `pages_about` is a prefix that we use to get to the about page. Instead of hard coding the actual link we use this prefix so that if the actual link changes the prefix will still work.

In your console type the following:

```
A styled app
/workspace/styled_app

$ rake routes
  Prefix Verb URI Pattern          Controller#Action
pages_about GET /pages/about(.:format) pages#about
  root GET /                  pages#home
```

You can see all the routes that are defined in the `routes.rb` file. The left column shows the prefix, then the type of http verb which for the two routes defined is a GET request. We then have the URI pattern column, This is how the link would look in the URL, it is the actual path to the resource. Then the final column should look somewhat familiar, it is the controller for the route and its action.

We can use the prefix along with the rails `link_to` helper method to create a link. All we do with the prefix is add `_path` to the end of it and the path to that resource gets generated for the anchor link. So to get to the about page we look at the prefix which is `pages_about` and append `_path` to then get `pages_about_path`. If we wanted to get our home page or root route we look at our home action and see the root prefix, we can then add `_path` to then get `root_path` which can again be used in combination with the rails helper method `link_to` to send us to the home page.

Excellent now we will add a link to the about page:

```
A styled app
/workspace/styled_app/app/views/pages/about.html.erb
```

```
<%= link_to 'Home', root_path %>
```

Remember the `root_path` helper comes from appending `_path` to the root prefix.

If you save and start up your server you should be able to navigate between the pages:

```
A styled app
/workspace/styled_app
```

```
$ rails server -b $IP -p $PORT
```

Shut the server down when you are done.

The Application Layout File

In the views folder above your pages folder you will see a layouts folder. Open it up and you will find the `application.html.erb` file. You can see that it looks like a normal html web page except for some stylesheet and javascript helper methods in the `<head>` tag.

Also in the body tag you will see a piece of embedded ruby code, It outputs something because of the `=` sign.

```
A styled app
/workspace/styled_app/app/views/layouts/application.html.erb
```

```
<!DOCTYPE html>
<html>
<head>
  <title>StyledApp</title>
  <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>

  <%= yield %>

</body>
</html>
```

The `yield` word means that it will show whatever view/template you are requesting. So when you visit the about page it is the `about.html.erb` view file that gets yielded.

Also note you are not passing any arguments to yield, You are just using the word yield on its own, In this situation yield renders the template or view of the controller and action that you are requesting. When we visit the about page we request the PagesController and in that the about action.

Why is this useful? It means you don't have to keep on typing out the doctype declaration and html and head tags etc. It remains in one place. If you look at your about.html.erb file you will see it has none of those tags, This is because you are essentially filling in the content for the body of the webpage. Look back at your application layout file and you will see yield is inside of the body tags. So the about template gets yielded when requested and again so does the home template.

All of this means you can easily add content that remains the same on every page, such as navigation. In your application.html.erb file add:

```
A styled app
/workspace/styled_app/app/views/layouts/application.html.erb

<!DOCTYPE html>
<html>
<head>
  <title>StyledApp</title>
  <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>
  <nav>
    <%= link_to 'HOME', root_path %>
    <%= link_to 'ABOUT', pages_about_path %>
  </nav>

  <%= yield %>

</body>
</html>
```

Add a nav tag and inside it add the same links. Remove the links from your home.html.erb and about.html.erb view templates:

```
A styled app
/workspace/styled_app/app/views/pages/about.html.erb
```

```
A styled app
/workspace/styled_app/app/views/pages/home.html.erb
```

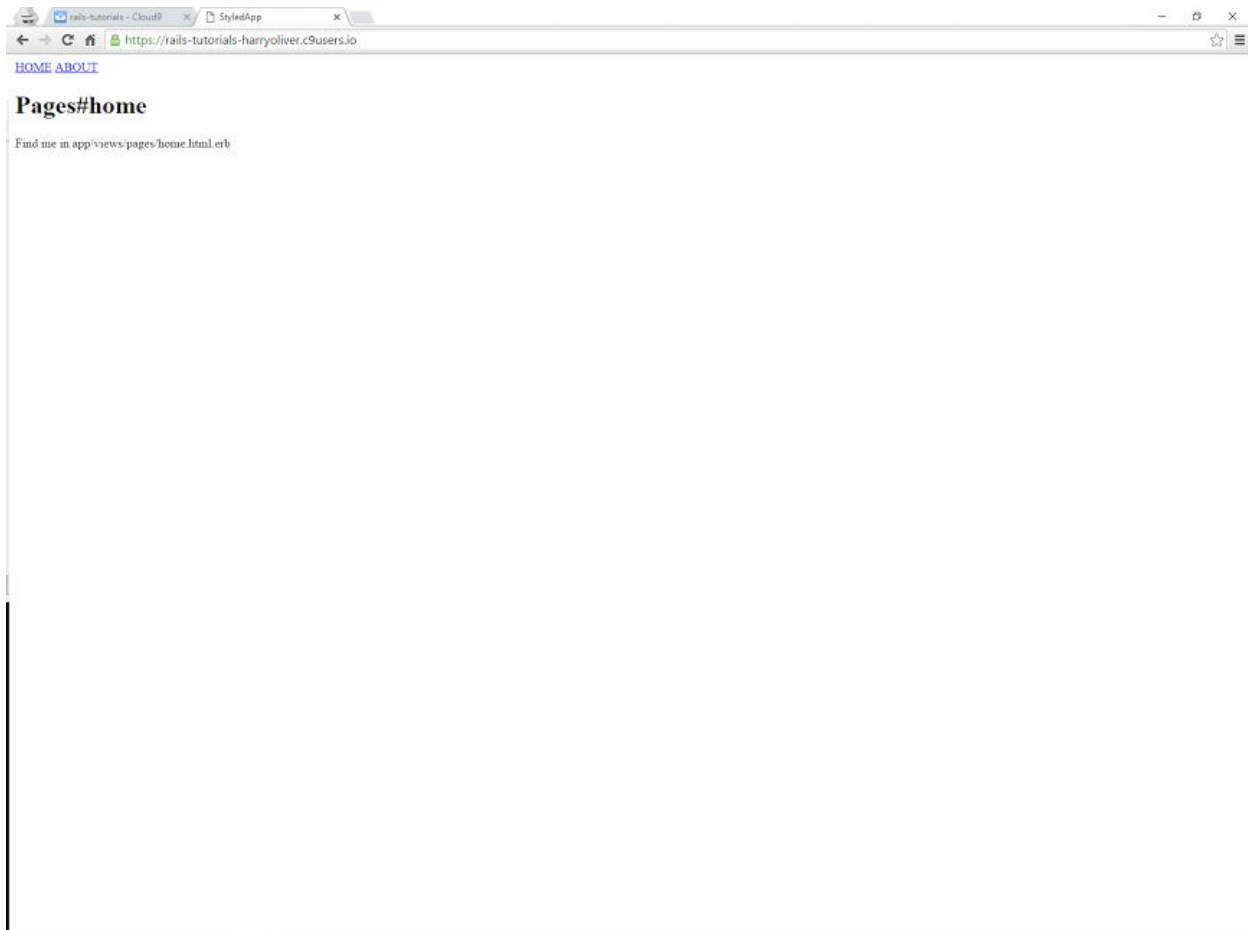
```
<h1>Pages#home</h1>
<p>Find me in app/views/pages/home.html.erb</p>
```

Remember to save and run your server:

```
A styled app
/workspace/styled_app
```

```
$ rails server -b $IP -p $PORT
```

No matter what page you are on you will see the links. This is because they are in the application layout file. They are not in a template that is being yielded so they remain there the whole time.



Let's add a little bit more to the layout file. As well as a nav tag we shall add a footer as well as most pages have a footer beneath the content.

The three dots represent code I am not showing, just to save space, Only add the highlighted code.

```
A styled app  
/workspace/styled_app/app/views/layouts/application.html.erb
```

```
<!DOCTYPE html>  
  
...  
  
  <nav>  
    <%= link_to 'HOME', root_path %>  
    <%= link_to 'ABOUT', pages_about_path %>  
  </nav>  
  
<%= yield %>  
  
  <footer>  
    <p>Copyright <%= Date.current.year %></p>  
  </footer>  
  
</body>  
</html>
```

As you can see we have a semantic (it describes itself) html footer tag and inside it is a paragraph tag. If you look at the paragraph tag there is the word copyright and then some embedded ruby tags, remember these tags allow us to use ruby code in our html view files. The = sign means the code gets outputted to the screen.

We use the Date method, we then call the rails current method on the date method which gives us the current time, We then use the year method on the current time and the result of these chained together methods is the current year. These methods may seem a bit confusing however they are all methods of the Date class. You can see and try out other methods [here](#) such as the yesterday method (I wonder what that will output) or the ago method which can be used to see how many seconds ago something was submitted for example.

Now that we have a navigation and a footer I think the home page and about page need sprucing up a bit. You can add what you want inside of the files or just copy me, it's up to you.

```
A styled app  
/workspace/styled_app/app/views/pages/home.html.erb
```

```
<h1>HOMELY</h1>  
<p>Welcome home</p>  
<p>Sometimes it is nice to just put your feet up and relax</p>  
<p>If you haven't done it recently then I would really recommend it!</p>
```

```
A styled app
/workspace/styled_app/app/views/pages/about.html.erb
```

```
<h1>ABOUT</h1>
<h3>Who am I?</h3>
<p>To you I am a stranger but to others I am a friend</p>
<p>Which one are you? Only I decide in the end.</p>
```

Okay now that the home and about pages have some content it's time to add a little bit of style to the page!

Styling The Pages

In the assets folder and in the stylesheets folder is a pages.scss stylesheet. This is where we will put our CSS to style the webpage.

```
A styled app
/workspace/styled_app/app/assets/stylesheets/pages.scss
```

```
// Place all the styles related to the Pages controller here.
// They will automatically be included in application.css.
// You can use Sass (SCSS) here: http://sass-lang.com/

* {
  margin: 0; //reset the padding and margin
  padding: 0;
}

body {
  font-family: sans-serif; //change the font of the webpage
  text-align: center; // center the text
}

nav {
  background: #222; //color the navigation black
  margin-bottom: 1em; //add margin at the bottom of nav for whitespace

  a { //style the links
    display: inline-block; //align the links next to each other
    padding: 1em 2em; //padd the links out make them bigger
    color: #fff; // color the text white
    text-decoration: none; // remove the underline from the links
  }
}

h1, h3 {
  margin: 1em 0; // add a top and bottom margin of 1 em to the headings, give left and right
  0 margin
}

p {
```



```
padding: .5em 0; // give the paragraphs some padding
}

footer { // style the footer
border-top: 1px solid #eee; // add a light grey border to the top of the footer
padding: 1em 0; // padd out the footer
font-style: italic; //change the font to italic
position: absolute; // change the position so the footer sticks to the bottom of the page
bottom: 0; // make the footer stick to the very bottom
left: 0; // center the text
right: 0; // center the text
}
```

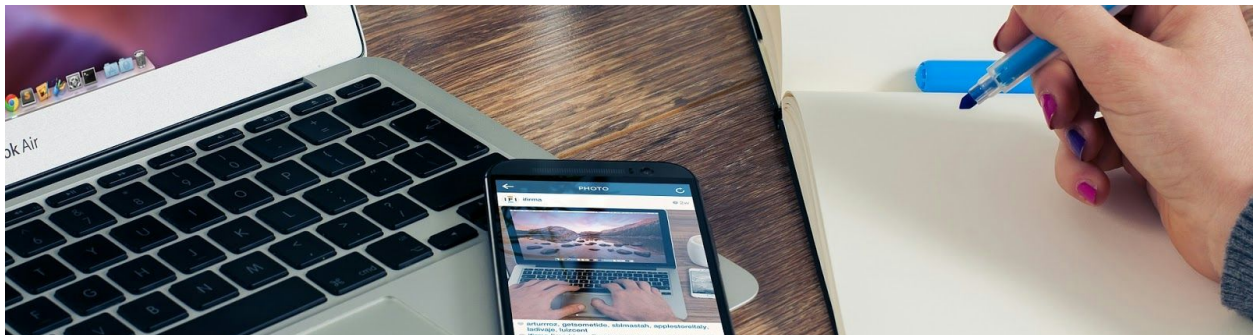
Feel free to copy and paste this css or add your own. In reality you wouldn't use this css in a website, you would use a css framework like bootstrap that adds some nice default styles we can use. However that will come later.

The text after the two forward slashes // is a css comment, you don't need to type them out, they are here just for a quick explanation.



Nicely done you have just built your second rails app, It was similar to the first app however you added an extra about action and view template. You then added a nav and a footer to the layout so that you could navigate between pages with the link_to rails helper method. Finally you styled the webpage with css.

WELL DONE!



Challenge: A Static Rails App

Okay it is time for a challenge, I will give you some instructions to follow and you will build your own rails app. What! I hear you say but I've just built two rails apps and your making me build yet another??!!? Yeah sorry, except this time you will be getting less help. It is a good way to learn, also you will quickly realize the things that you don't know. When doing this exercise remember to double check everything you type and don't worry if you muck up as you can just start over. When I first started out some simple things I got stuck on were not adding a comma in the correct place on a link_to helper method or forgetting to use an = sign when using embedded ruby so nothing got output to the screen, Essentially just syntax errors, Okay with that said feel free to look back at the previous chapters for help.

Step 1

CD into your **workspace** directory

Step 2

Generate a new rails app called my_pages (You don't need to use a specific version number)

Step 3

CD into the new rails app (**my_pages**)

Step 4

Run the server and check that you get the default rails page

Step 5

Generate a controller named **Pages** and at the same time generate a **home** and **about** action (You should get the views generated as well etc.)

Step 6

Open your **routes.rb** file and change the get home route to a **root route**, (remember the # sign syntax)

Step 7

Run the server and check you are directed to the home page

Step 8

Open the **application.html.erb** file and add a nav html tag. Inside the nav tag add two rails links:

One link for the Home page and one link for the About page.

(Remember you can: **rake routes**)

Step 9

Update the **home.html.erb** file and **about.html.erb** file, add your own content.

Step 10

Open the **pages.scss** stylesheet and add some styles to your app.

Success In 10 steps you have made your own rails app with no help, pretty good going if you ask me. What did you get stuck on? Was it remembering certain rails commands or maybe some specific syntax? Make a note of it and stick it to your screen!

A Simple Form App

The next rails app will be a very simple form, you will generate a model (a class that defines attributes) with a name attribute, and then submit the data to the database. You will then show all the names on a page. There are a few new things to learn in this lesson, however it should soon click together.

Make sure you are in the workspace directory:

```
~/workspace $ pwd
/home/ubuntu/workspace
```

Generate a new rails app called form_app

```
A simple form app
/workspace
$ rails 4.2.5_ new form_app
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  create  app/helpers/application_helper.rb
  create  app/views/layouts/application.html.erb
  create  app/assets/images/.keep
  create  app/mailers/.keep
  ...
```

Change directory into the new app:

```
$ cd form_app/
~/workspace/form_app $
```

Run the server: Check all is fine.

```
A simple form app
```

```
/workspace/form_app
```

```
$ rails server -b $IP -p $PORT
```

Generate a people controller:

```
A simple form app  
/workspace/form_app
```

```
$ rails generate controller People index new  
Running via Spring preloader in process 726  
  create  app/controllers/people_controller.rb  
  route   get 'people/new'  
  route   get 'people/index'  
  invoke  erb  
  create  app/views/people  
  create  app/views/people/index.html.erb  
  create  app/views/people/new.html.erb  
  invoke  test_unit  
  create  test/controllers/people_controller_test.rb  
  invoke  helper
```

Here we are generating a controller called People, This is because in this app we will be submitting people's names so a controller of people seems appropriate. If you remember we also get the index.html.erb file and the new.html.erb file generated for us, (as we are specifying those two actions in the command) and we also get some routes automatically generated in the routes.rb file.

Update the routes

```
A simple form app  
/workspace/form_app/config/routes.rb
```

```
Rails.application.routes.draw do  
  
  resources :people  
  root 'people#index'  
  
  # The priority is based upon order of creation: first created -> highest priority.  
  # See how all your routes lay out with "rake routes".  
  
  # You can have the root of your site routed with "root"  
  # root 'welcome#index'
```

Change the index action so that it is the root route, when you load the application this will be the first page that shows. The resources line automatically creates many routes for us. Use **rake routes** to take a quick look. (We will cover them more later).

Run the server:

```
A simple form app
/workspace/form_app

$ rails server -b $IP -p $PORT
```

You should see the index.html.erb page, if you add /people/new to the end of the url you will also see the new page. As you might have guessed this will be the page where we create a new person.

Generate the person model

```
A simple form app
/workspace/form_app

$ rails generate model Person
Running via Spring preloader in process 875
  invoke  active_record
  create  db/migrate/20160226194318_create_people.rb
  create  app/models/person.rb
  invoke  test_unit
  create  test/models/person_test.rb
  create  test/fixtures/people.yml
```

Here we are generating a **model**, we have named the model Person. A model is used to define an object. What do I mean by object? An object can be anything, in this case we are defining a single person. The model file can be found under app/models/person.rb if you open it up you will see that it is empty. So a model file of person.rb got generated. If you look you will see a migration file got generated under db/migrate/20160226194318_create_people.rb, that number is a timestamp, yours will be different.

The migration file is the file that updates the database, Even though you have a person.rb model file your application does not actually have anywhere to save the person into because we have not updated the database. So the migration file is used to update the database.

Look closely, you used the rails generate model command to generate a Person model, the model file that got generated was indeed called person.rb, but look at the migration file, db/migrate/20160226194318_create_people.rb the file is called create_**people**.rb so it got pluralized to people. The migration file creates the table in the database. It updates the database with the information that we will put in it. Since it is a table that holds information on not one person but many people this pluralization starts to make a little more sense. This is one of rails conventions, A model will be singular (person) but a database table will be plural (people as it holds information on many people). Don't worry about memorizing this initially, I will remind you throughout the book.

```
A simple form app
/workspace/form_app/app/models/person.rb
```

```
class Person < ActiveRecord::Base
end
```

Even though your person.rb file is empty it is still used by rails to generate a person object and save it to the database. We will see later how we can use this file to validate attributes. For example if you wanted people to only enter a name longer than 3 characters.

Updating the migration file

```
A simple form app
/workspace/form_app/db/migrate/20160226194318_create_people.rb
```

```
class CreatePeople < ActiveRecord::Migration
  def change
    create_table :people do |t|

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

The migration file has a class of CreatePeople, which is just a normal ruby class. This class inherits from ActiveRecord::Migration which is part of rails. This allows us to then use this file to update our database.

Inside the CreatePeople class we have a method (defined with the def and end keywords) called change. The change method has inside it something called a schema statement. The create_table statement is a statement that is used to create a table called people.

The create_table statement takes a parameter for the name. **:people** is the name of the table and it has a colon in front of it. why? It is called a symbol, If you have never come accross a symbol before then know that it is similar to a string except that it uses far less memory. It is used to represent something, in this case the name of the table.

After the symbol we have a **do** keyword and **end** keyword a few lines down. This is what you call a block. A block is code that is used by other code. If you look just after the do keyword you see **|t|** This is a temporary variable that is used to specify the columns we will add to the table. You can see this with t.timestamps which creates created_at and updated_at columns, (don't worry too much about that for now though).

Add the following line to your migration file.

This line creates a column in the people table, the column is called name and it is of type string. The type of string means that we will be putting short content in there, less than 255 characters.

```
A simple form app
/workspace/form_app/db/migrate/20160226194318_create_people.rb
```

```
class CreatePeople < ActiveRecord::Migration
  def change
    create_table :people do |t|
      t.string :name
      t.timestamps null: false
    end
  end
end
```

Update The Database

```
A simple form app
/workspace/form_app/
```

```
$ rake db:migrate
== 20160226194318 CreatePeople: migrating =====
-- create_table(:people)
   -> 0.0012s
== 20160226194318 CreatePeople: migrated (0.0013s) =====
```

This command runs the migration file that you just updated. As you can see by the output a table gets created called people.

If you look in the db folder you will see a new file called schema.rb, The database schema has been updated and if you open up the file you will see the table you have created.

```
A simple form app
/workspace/form_app/db/schema.rb
```

```
...
ActiveRecord::Schema.define(version: 20160226194318) do

  create_table "people", force: :cascade do |t|
    t.string "name"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

end
```

You can see the person table has a column called name which is of type string. You can also see that the timestamps line in the migration file created two columns, one of created_at and one of updated_at. These can be used to see when an attribute was created, A good example is a blog post gets created_at a certain time.

Okay you have created a person model file, you have created and updated your migration file and you have also migrated your database schema to update it with your people table. What next?

Update The PeopleController

```
A simple form app  
/workspace/form_app/app/controllers/people_controller.rb
```

```
class PeopleController < ApplicationController  
  def index  
  end  
  
  def new  
    @person = Person.new  
  end  
end
```

The text @person is called an instance variable. It is available to any method that you write in the person class (person.rb file). I recommend you read up a little on instance variables if you are shaky on them. In rails, instance variables in the controller are made available to the view files.

Here we create a new person object/instance with the line Person.new (capital P). This is using our person.rb file to create a person object, which then gets saved in the instance variable @person. The = sign is the assignment operator, the new Person object is stored/assigned to the @person variable. Now we will use this @person variable in our view file.

Add A Form To The new.html.erb View

```
A simple form app  
/workspace/form_app/app/views/people/new.html.erb
```

```
<%= form_for @person do |f| %>  
  <%= f.text_field :name, :placeholder => "Name" %>  
  <%= f.submit "Submit" %>  
<% end %>
```

In this view file we use erb or embedded ruby to create the form. We use an = sign on the first line to output the form, then the form_for method which uses the @person instance variable that we defined in our controller new action. So in the view we are now using that new Person object

that we have created. We then create a block with the do keyword and a temporary variable called |f| which is used to create the form fields.

The top line of erb is then closed, if you look at the bottom you will see <% end %> which closes out our block of ruby code. Notice it doesn't have an equals sign since we don't want to output that code.

In Between our block of code we have more embedded ruby that outputs stuff. we use the temporary variable to create a text_field which relates to our name field in the people table. (Go open up the schema.rb file again and see that the people table has a name column so we are creating a name text_field so that we can save a name to the database) Also you will see a placeholder, this puts text inside the text field automatically so as to prompt the user. The other erb line is f.submit which creates a submit button so that we can submit our data to the database. The line "Submit" is what will show on the button.

Run the server:

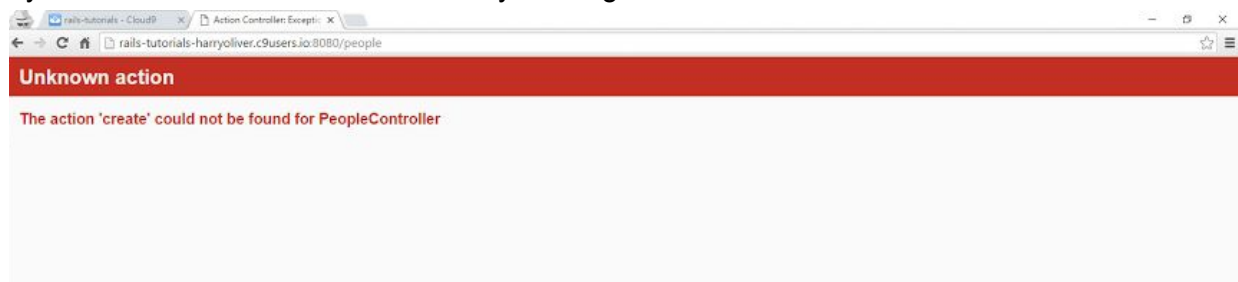
```
A simple form app
/workspace/form_app

$ rails server -b $IP -p $PORT
```

Navigate to /people/new and you will see a name text_field and a submit button that looks like this:



If you click the submit button however you will get an error:



The error says the create action could not be found. The form is using the new action in the people controller and then looks for an action called create when the submit button is pressed.

Adding The Create Action

```
A simple form app  
/workspace/form_app/app/controllers/people_controller.rb
```

```
class PeopleController < ApplicationController  
  def index  
    end  
  
  def new  
    @person = Person.new  
    end  
  
  def create  
    @person = Person.create(person_params)  
    if @person.save  
      redirect_to '/'  
    else  
      render :new  
    end  
  end  
  
  private  
  def person_params  
    params.require(:person).permit(:name)  
  end  
end
```

We have added a create action and a person_params method which is private as it is underneath the private keyword. This means that the method person_params cannot be called with a receiver.

The idea is that private methods are hidden, and can only be called in their own class. In this case the Person class. I recommend you google about ruby private methods as it is a bit confusing.

In rails an action renders a template, the new action renders the new.html.erb file. So we hide the person_params method to ensure rails doesn't route any HTTP calls to that action.

The person_params method uses the line params.require(:person).permit(:name) to whitelist what we submit to our database. Here we say only allow the :name to be submitted.

If we look back at the create action now we are creating a new Person with the create method, and we are passing the person_params method to the create method, so that when we create a new person we only allow the name. We then store this in the @person instance variable. We have an if statement that checks is the @person we created got saved to the database. If this is true and it got saved to the database then we redirect to the index template.

The create method automatically creates an object (a person object) and tries to save it to the database, If the create method cannot do this then the else statement that checks whether the @person got saved will run and we will render the new page again.

Run the server:

```
A simple form app
/workspace/form_app

$ rails server -b $IP -p $PORT
```

Navigate to /people/new and add some names and submit them, do this a couple of times.

Updating The Index.html.erb Page And Index Action

If you look at your index page you will see that it isn't showing any of the names we have submitted to the database. Time to fix that.

```
A simple form app
/workspace/form_app/app/controllers/people_controller.rb

class PeopleController < ApplicationController
  def index
    @people = Person.all
  end

  def new
    @person = Person.new
  end

  def create
    @person = Person.create(person_params)
    if @person.save
      redirect_to '/'
    else
      render :new
    end
  end

  private
  def person_params
    params.require(:person).permit(:name)
  end
end
```

In the people controller index action we create an instance variable called @people because we want to store every person in our database in that instance variable so we pluralize the name (despite the fact it could be called anything). We then target our Person class and call the .all

method to query all the people in the database. We can now use the instance variable in our index.html.erb view.

```
A simple form app
/workspace/form_app/app/views/people/index.html.erb
```

```
<% @people.each do |p| %>
  <p>Name: <%= p.name %></p>
<% end %>
<%= link_to "Add Name", new_person_path %>
```

In the index file add the embedded ruby above, We take the instance variable `@people` from our controller index action and call the ruby method `each` on it. This iterates through every person in the database. We create a paragraph with the text `name:` and then using a temporary variable `|p|` and `erb` we print out the name of each person object. The name comes from the `name` attribute in our database. If you open the `schema.rb` file and look at the `people` table you will see the `name` column, This is what we are targeting for each person object that we loop through with the above code. Remember we use an equals sign in `erb` to output to the screen, Since we only want to print out the name we only use `erb` with an equals sign in the paragraph tag. You can add an `=` sign to the first line of `erb` if you wish, try it and see what happens. Then remove it when you are done. (In order for anything to show up make sure you have added some names to your database).

Run the server:

```
A simple form app
/workspace/form_app

$ rails server -b $IP -p $PORT
```

Try adding some names, Each time you do you will be redirected to the index page. If you add a blank name then that will show up as blank. We will cover validations in the next project so that you can make sure a name is a certain length before it is saved to the database.



The index page shows names that have been submitted as well as blank names.

That was a lot, nicely done! You created a very simple form with one form field for people to put their names and then submit their names to the database. You then collected all those names and looped through them and displayed them on the index page. Nice Job!

WELL DONE!



A Styled Form App

In this project you will build a form for a user to submit an email and name. You will learn how to add validations to the model file so that a name for example, has to be a minimum length before being submitted. You will also learn how to add gems to the gemfile and install them, as well as configuring them. The gem you will learn about is bootstrap, and is used for styling your app.

Make sure you are in the workspace directory:

```
~/workspace $ pwd
/home/ubuntu/workspace
```

Generate a new rails app called styled_form

```
A styled form
/workspace

$ rails _4.2.5_ new styled_form
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  create  app/helpers/application_helper.rb
  create  app/views/layouts/application.html.erb
  create  app/assets/images/.keep
  create  app/mailers/.keep
  ...
```


Change directory into the new app:

```
$ cd styled_form/  
~/workspace/styled_form $
```

Run the server: Check all is fine.

```
A styled form  
/workspace/styled_form
```

```
$ rails server -b $IP -p $PORT
```

Generate a User model:

```
A styled form  
/workspace/styled_form
```

```
$ rails generate model User  
Running via Spring preloader in process 718  
  invoke  active_record  
  create  db/migrate/20160301193656_create_users.rb  
  create  app/models/user.rb  
  invoke  test_unit  
  create  test/models/user_test.rb  
  create  test/fixtures/users.yml
```

Update the migration file:

```
A styled form  
/workspace/styled_form/db/migrate/20160301193656_create_users.rb
```

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

Add a name column and an email column and make the, both of type string.

Migrate the database:

```
A styled form
/workspace/styled_form
```

```
$ rake db:migrate
== 20160301193656 CreateUsers: migrating =====
-- create_table(:users)
   -> 0.0024s
== 20160301193656 CreateUsers: migrated (0.0026s) =====
```

Migrate your database and create the Users table. You can see the result in the `db/schema.rb` file.

Generate a Users controller

```
A styled form
/workspace/styled_form
```

```
$ rails generate controller Users index new
Running via Spring preloader in process 789
  create  app/controllers/users_controller.rb
  route  get 'users/new'
  route  get 'users/index'
  invoke  erb
  create  app/views/users
  create  app/views/users/index.html.erb
  create  app/views/users/new.html.erb
  invoke  test_unit
  create  test/controllers/users_controller_test.rb
  invoke  helper
  create  app/helpers/users_helper.rb
  invoke  test_unit
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/users.coffee
  invoke  scss
  create  app/assets/stylesheets/users.scss
```

The generate controller command creates a `users_controller.rb` file with the actions `index` and `new`. Open the file up to take a look. If you are wondering why didn't I add the `create` action to the list it is because a `create.html.erb` view file would also get generated and we would end up deleting it. (A rails action by default renders a view).

Update the routes:

```
A styled form
/workspace/styled_form/config/routes.rb
```

```
Rails.application.routes.draw do
```

```
  resources :users  
  root 'users#index'
```

```
  ...  
end
```

Update the routes in your routes.rb file. Setup the root route and add the resources line. Remember you can see the routes resources generates by using the rake routes command.

Run the server:

```
A styled form  
/workspace/styled_form
```

```
$ rails server -b $IP -p $PORT
```

Check the links work by loading up the server and then navigate to /users/new as well.

Update the users_controller file:

```
A styled form  
/workspace/styled_form/app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController  
  def index  
    end  
  
  def new  
    @user = User.new  
    end  
  
  def create  
    @user = User.create(user_params)  
    if @user.save  
      flash[:success] = "You Signed Up woo!"  
      redirect_to '/'  
    else  
      flash[:danger] = "There was an error try again!"  
      render :new  
    end  
  end  
  
  private  
  
  def user_params  
    params.require(:user).permit(:name, :email)  
  end  
end
```

In the new action you create a new user by calling the new method on your User class. You then store the new User object in the @user instance variable.
In the create action you create a User with the permitted params which are in the user_params private method. In the method you allow only the name and email parameters.
If the user is saved then you redirect to the root page and if there was an error you render the new action again.

If the user was saved then we set a flash hash, this allows us to show a message in the next action that gets rendered. With the flash hash we can show a message to the user and let them know if they were successful or not.

The key or name of the flash is :success and the value or message is the string "You Signed Up Woo" This message is what we display to the user. The key or name of the hash is what we use for styling/identifying.

If the User is not saved to the database then we render the new action and try again. However we also display a different flash hash with a new message.

The hash has a key or name of :danger and a value or message of "There was an error try again!" The message will get displayed to the user if there was a problem.

If you are wondering can the key or name of the hash be anything? The answer is yes, just like the value or message you can give it any name you want, The reason I chose :success and :danger is because these are two styles used in bootstrap.

Show The Flash Messages:

```
A styled form
/workspace/styled_form/app/views/layouts/application.html.erb

<!DOCTYPE html>
<html>
<head>
  <title>StyledForm</title>
  <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>

  <% flash.each do |name, message| %>
    <%= content_tag :div, message, class: "alert alert-#{name}" %>
  <% end%>

  <%= yield %>

</body>
</html>
```

We use some embedded ruby in the layout file to iterate through each flash message (if there is more than one) and show its content. We target the flash hash and call the each method on it to go through each flash hash.

In ruby a hash has a key and a value. We set the key and value back in the controller file. The key was :success and the value was a string.

Here the key is name and the value is message. We have used variables to represent the key and value of the hash. They could be called anything. We could have called the variables |k, v| for key and value, but it is nicer to be a bit more descriptive.

We then output (notice the erb with an equals sign, the other erb is logic code that we don't want to output) a div tag, Rails has a helper method called content_tag which allows us to specify a html tag that we can then add content to.

The message in the div that gets displayed is the variable called message, This message variable is set to the string from the controller file. After the comma you see the class: word, This is the rails way of adding a class to a html element. As we are generating the div tag with the helper method we can't add the class in the normal way like with html class="class-name" We use the rails syntax of class: with a colon after it.

The div gets given a class of alert and a class of alert-name, the name is actually the variable just like message is also a variable. The name variable will either have a value of success or danger because of how we named our flash hashes in the controller.

Just after the dash is a pound sign followed by two curly braces, This is interpolation and converts the name variable into its value, so either success or danger.

The class then becomes "alert alert-success" not "alert alert-name". If we didn't use string interpolation to convert the variable then the class would remain as alert-name which is not what we want.

You don't have to do it this way, you could just have a static class of notice if you like, however this is a bit nicer for the person viewing the site.

Update the new.html.erb file with a form:

```
A styled form
/workspace/styled_form/app/views/users/new.html.erb

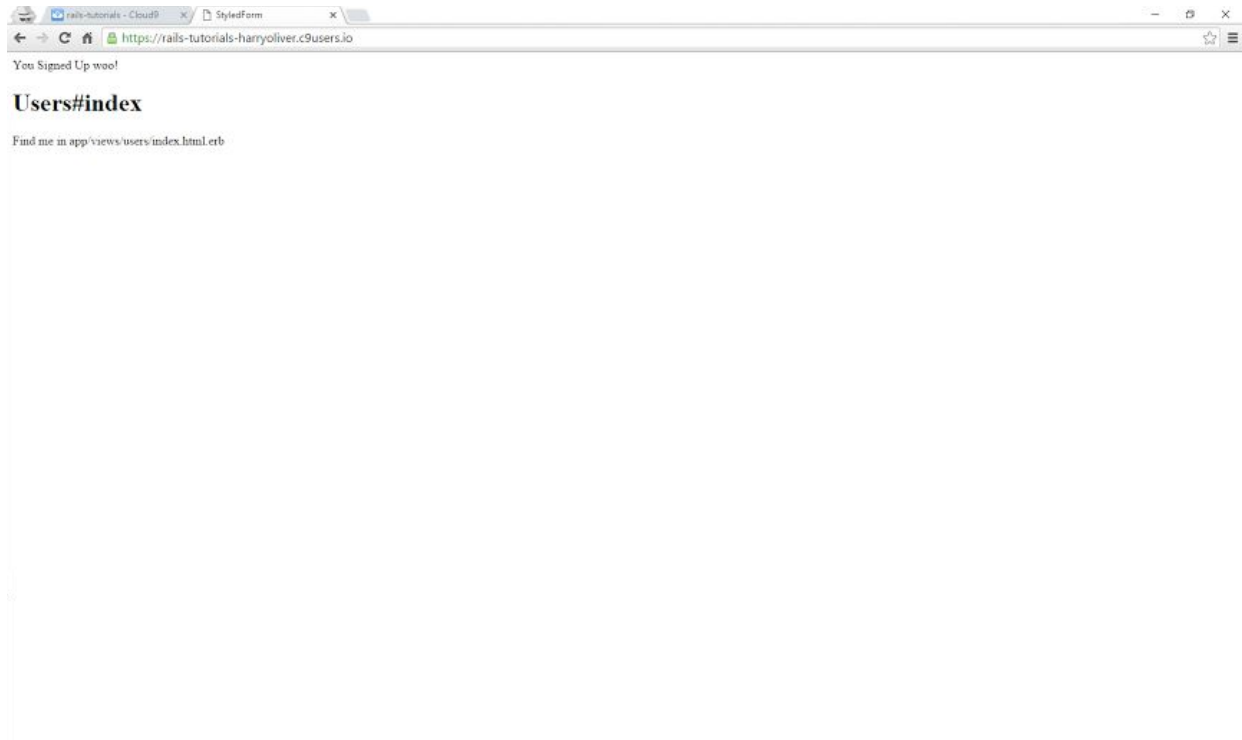
<%= form_for @user do |f| %>
  <%= f.text_field :name, :placeholder => "Name" %>
  <%= f.email_field :email, :placeholder => "Email" %>
  <%= f.submit "Submit" %>
<% end %>
```

Run the server:

```
A styled form
/workspace/styled_form
```

```
$ rails server -b $IP -p $PORT
```

If you navigate to /users/new you should see a form, Enter some details and submit them. You will be redirected to the index page, You will also see the flash message of success. Pretty cool!



Validations

```
A styled form
/workspace/styled_form/app/models/user.rb
```

```
class User < ActiveRecord::Base
  validates :name, presence: true
  validates :email, presence: true
  validates :name, length: {minimum: 3}
  validates :email, length: {minimum: 5}
end
```

Add the following lines to your user.rb model file. These are the validations that I talked about earlier. The validates helper method takes a symbol of the attribute you are looking to validate. The users table has the attributes name and email so these are what we target.

Here we validate the presence and set it to true to make sure that something has been typed into the input boxes. We also validate the length of the attributes and set them a minimum constraint.

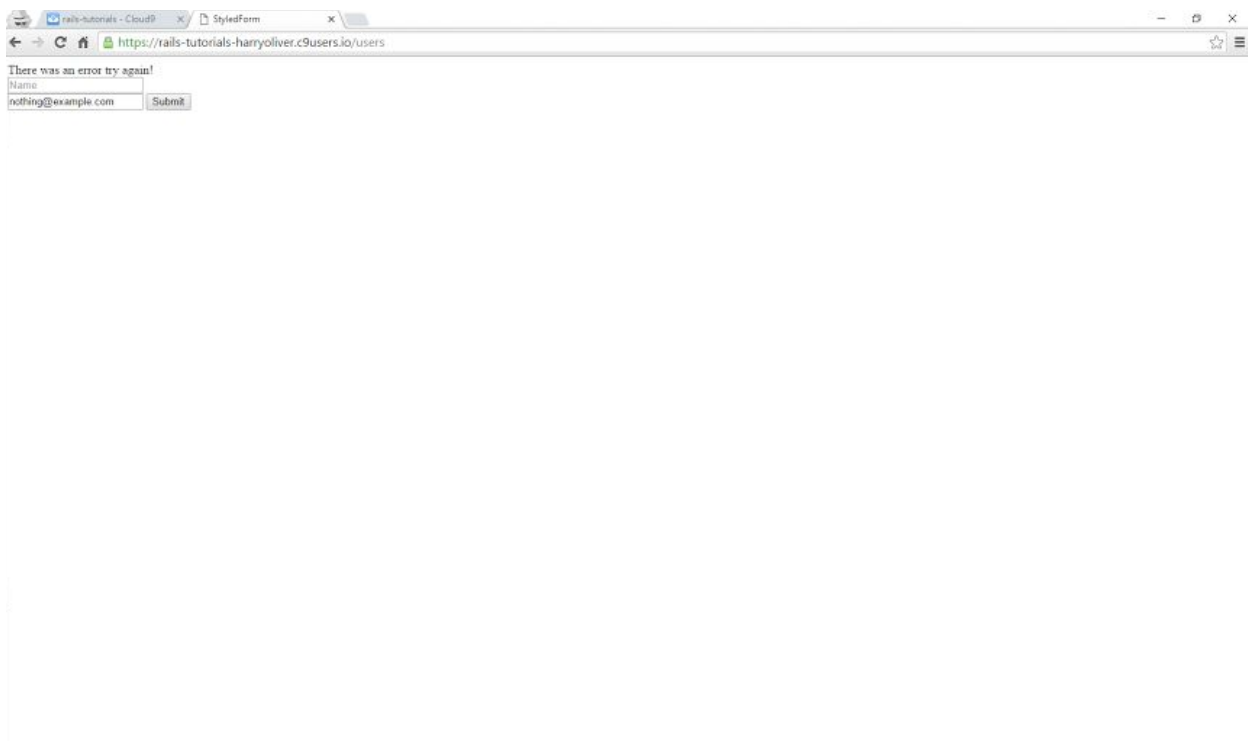
The name must be at minimum 3 characters and we make sure the email is also a minimum of 5 characters. These are some simple validations, we will cover some others later on.

If you enter some information into the input fields now and they are blank or too short. then you will get a flash error message that you set up in the controller and application.html.erb files and you will be redirected to the new.html.erb page to try again.

Run the server:

```
A styled form  
/workspace/styled_form
```

```
$ rails server -b $IP -p $PORT
```



Add The Bootstrap Gem:

```
A styled form  
/workspace/styled_form/app/gemfile
```

```
source 'https://rubygems.org'
```

```
# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.5'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.1.0'
# See https://github.com/rails/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby
gem 'bootstrap-sass', '~> 3.3.6'
...
```

The gem file is the location of all your pre installed gems (pre packaged code) They allow you to quickly add features to your app without having to reinvent the wheel and do everything from scratch. For example if you need to upload attachments or images there is a gem for that called paperclip, all you do is install it and set it up and off you go. Configuration of gems will vary from gem to gem however you can find simple tutorials on their associated github pages.

I have excluded some of the gemfile as it is quite large. Add the gem bootstrap-sass line.

Bundle Install

```
A styled form
/workspace/styled_form

$ bundle install
Fetching gem metadata from https://rubygems.org/.....
Fetching version metadata from https://rubygems.org/...
Fetching dependency metadata from https://rubygems.org/..
Resolving dependencies...
...
```

The bundle install command installs all the gems in the gemfile (I have shortened the output). Since the bootstrap gem was added to the file the bundle install command has to be run. This fetches the gems and any dependencies and installs them for you to use in your application. After that you just need to do some configuration.

Rename The application.css File To application.scss

```
A styled form
/workspace/styled_form/app/assets/stylesheets/application.css

$ mv app/assets/stylesheets/application.css app/assets/stylesheets/application.scss
```


Here I use the move command to rename the file and change the extension from **css** to **scss**. That is all we are changing. You can just right click on the file and click rename if you prefer.

Import Bootstrap Styles:

```
A styled form  
/workspace/styled_form/app/assets/stylesheets/application.scss
```

```
/*  
 * This is a manifest file that'll be compiled into application.css, which will include all  
 the files  
 * listed below.  
 *  
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets,  
 vendor/assets/stylesheets,  
 * or any plugin's vendor/assets/stylesheets directory can be referenced here using a  
 relative path.  
 *  
 * You're free to add application-wide styles to this file and they'll appear at the bottom  
 of the  
 * compiled file so the styles you add here take precedence over styles defined in any  
 styles  
 * defined in the other CSS/SCSS files in this directory. It is generally better to create a  
 new  
 * file per style scope.  
 *  
 *= require_tree .  
 *= require_self  
 */
```

Your file will look like the one above, Make the changes shown below:

```
A styled form  
/workspace/styled_form/app/assets/stylesheets/application.scss
```

```
/*  
 * This is a manifest file that'll be compiled into application.css, which will include all  
 the files  
 * listed below.  
 *  
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets,  
 vendor/assets/stylesheets,  
 * or any plugin's vendor/assets/stylesheets directory can be referenced here using a  
 relative path.  
 *  
 * You're free to add application-wide styles to this file and they'll appear at the bottom  
 of the  
 * compiled file so the styles you add here take precedence over styles defined in any  
 styles  
 * defined in the other CSS/SCSS files in this directory. It is generally better to create a  
 new  
 * file per style scope.  
 *
```

```
*/  
@import "bootstrap-sprockets";  
@import "bootstrap";
```

Remove the `require_self` and `require_tree` lines and add the `@import` lines beneath.

Require Bootstrap Javascripts:

```
A styled form  
/workspace/styled_form/app/assets/javascripts/application.js  
  
// This is a manifest file that'll be compiled into application.js, which will include all  
// the files  
// listed below.  
//  
// Any JavaScript/Coffee file within this directory, lib/assets/javascripts,  
// vendor/assets/javascripts,  
// or any plugin's vendor/assets/javascripts directory can be referenced here using a  
// relative path.  
//  
// It's not advisable to add code directly here, but if you do, it'll appear at the bottom  
// of the  
// compiled file.  
//  
// Read Sprockets README (https://github.com/rails/sprockets#sprockets-directives) for  
// details  
// about supported directives.  
//  
//= require jquery  
//= require jquery_ujs  
//= require bootstrap-sprockets  
//= require turbolinks  
//= require_tree .
```

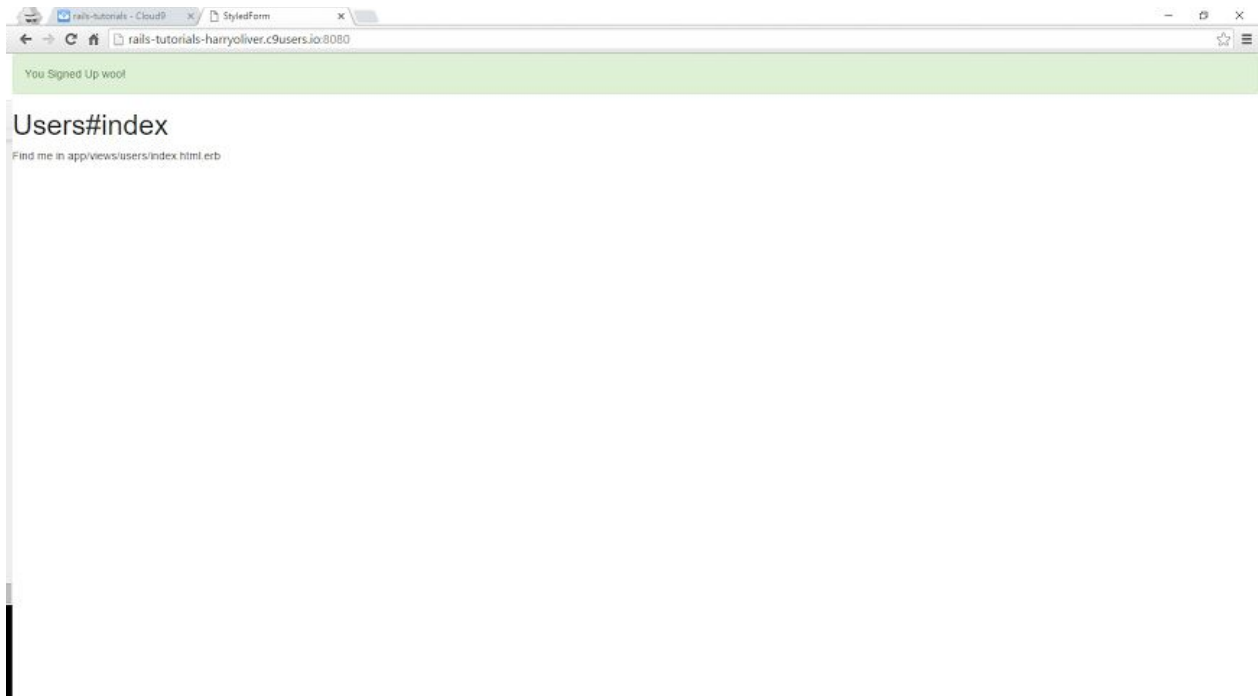
You only need to add one line to the `application.js` file. Make sure you add it below the other two `jquery` lines.

Run the server:

```
A styled form  
/workspace/styled_form  
  
$ rails server -b $IP -p $PORT
```

Load up the server and you should see some slightly different styling such as a sans-serif font. You will also see that the flash messages have been styled with bootstrap. That is because we named the class `alert-success` and `alert-danger` which are bootstrap styles.

By adding a gem and installing it and then doing a small amount of configuration for the app we get some pretty nice styles.



Style The Index Page:

Update the index.html.erb view file and add a link to the new.html.erb page.

```
A styled form
/workspace/styled_form/app/views/users/index.html.erb

<div class="container">
  <div class="row">
    <div class="col-md-6 col-md-offset-3 text-center">

      <%= link_to 'SIGN UP', new_user_path, class: 'btn btn-success' %>

    </div>
  </div>
</div>
```

Here we have styled the index page with bootstrap classes. Bootstrap classes are designed so that you don't need to write your own utility classes etc. If you need to center some text then there is a bootstrap class for that. The framework helps to speed up the styling of your application. Of course if you want to override a style then all you need to do is add your own class and then add the rules to that css class in your stylesheet.

You will see the first html tag is a div and it has a class of container, This is a bootstrap class and it is required if we are adding a row class later on, All rows must go in a container so that they are formatted properly.

The next html tag is another div and has a class of row, The row is the width of the screen and can be broken down into 12 columns. This is the amount bootstrap has specified. This is know as a grid system and you can read more about it on the bootstrap website.

The next html div has a few more bootstrap classes, the class text-center is quite self explanatory, If you remove it and reload your page you will see the text is no longer centered. On the left is a class of col-md-6 which stands for a column that is 6 wide on a medium device. The md means medium. Don't worry too much about the md part, It allows you to style your site depending on what devices you think your site will be viewed on. So if you were targeting mobile users then you might use col-sm-6 sm stands for small. Check out these bootstrap [grid examples](#).

Remember I said the grid is made up of 12 columns. The current div is 6 columns wide but it will be on the left side of the page unless we offset it with a class. When i say offset I mean push over. We add the class col-md-offset-6 notice that there is an extra word (offset) and that it is only offset by 3. This is because it adds 3 to the left and right. So 6 wide plus 3 on the left and 3 on the right means a 12 column grid. Try adjusting the classes and removing the offset class to see what happens.

If you look at the rails link_to method you can see we add a class the rails way because we are generating this link with rails, It is not plain html so we change the syntax. (**class: 'class-name'**) The link gets given a bootstrap class of btn which stands for button and then another bootstrap class of btn-success which styles the button a green colour. Try changing the class to btn-primary and see what happens.

So in very few lines you have styled a link/anchor into a button and centered it on the page. If you were to write those css rules yourself it would have taken significantly more code.



Style The New Page:

Update the new.html.erb view file and style with bootstrap classes.

A styled form
/workspace/styled_form/app/views/users/new.html.erb

```
<div class="container">
  <div class="row">
    <div class="col-md-6 col-md-offset-3">

      <%= form_for @user do |f| %>

        <div class="form-group">
          <%= f.text_field :name, :placeholder => "Name", class: 'form-control' %>
        </div>

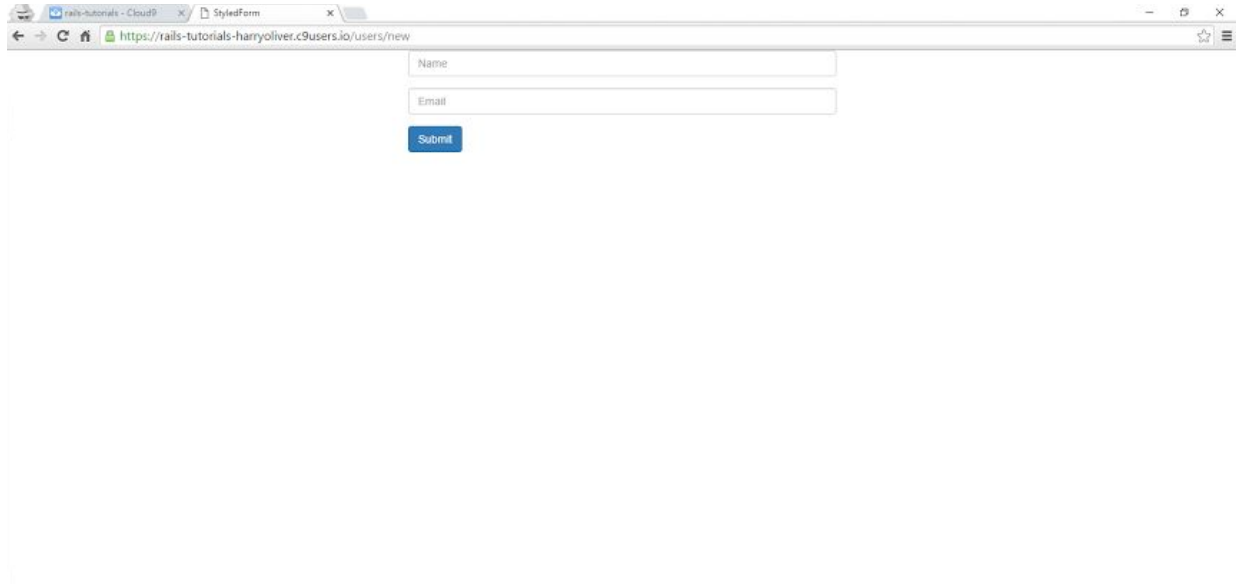
        <div class="form-group">
          <%= f.email_field :email, :placeholder => "Email", class: 'form-control' %>
        </div>

        <%= f.submit "Submit", class: 'btn btn-primary' %>
      <% end %>

    </div>
  </div>
</div>
```

First 3 divs are added with a container class, a row class which has to be inside a container class, and a div with column classes and offset classes so as to center the form on the page.

Then in the form two divs are added, they each wrap around an input field. They have a bootstrap class of form-group, This class adds some spacing and formatting so that the input boxes are not squashed up next to each other. Then each input field gets given a class of form-control. This increases the width of the input field, adds some styling such as padding and border-radius and makes the inputs dynamic so that they adjust to the size of the screen. The submit button also gets given a class of btn and btn-primary to style it blue,



Excellent you have created a form with model validations that check whether anything is present in the input boxes and also make sure that the input is a minimum length, If not a nicely styled error message is displayed. You also installed the bootstrap gem and configured it for use in your app. You then styled the pages with the bootstrap classes.

WELL DONE!



Challenge: A Bootstrap Rails App

In this challenge you will create a new rails app and install the twitter bootstrap gem. You will then configure the gem so that the bootstrap styles can be used in your app.

Step 1

CD into your **workspace** directory

Step 2

Generate a new rails app called `bootstrap_app` (You don't need to use a specific version number)

Step 3

CD into the new rails app (**`bootstrap_app`**)

Step 4

Run the server and check that you get the default rails page

Step 5

Generate a **Pages controller**, index action and `index.html.erb` view file

Step 6

Open your **routes.rb** file and change the get index route to a **root route**, (remember the # sign syntax)

Step 7

Run the server and check you are directed to the home page

Step 8

Add the bootstrap gem to the gemfile

```
gem 'bootstrap-sass', '~> 3.3.6'
```

Step 9

Install the gem with the command:

```
bundle install
```

Step 10

Rename your **application.css** file to have a **.scss** extension.

```
app/assets/stylesheets/application.css
```

```
app/assets/stylesheets/application.scss
```


Step 11

In your application.scss file add the @import lines at the bottom of the file

```
@import "bootstrap-sprockets";  
@import "bootstrap";
```

Step 12

Remove the require tree lines from the application.scss file

```
*= require_tree .  
*= require_self
```

Step 13

Add the require bootstrap-sprockets line to your application.js file (add it below the jquery lines)

```
//= require bootstrap-sprockets
```

Step 14

Add a link to your index page that just links to the root_path, Style the link into a button with bootstrap classes. (btn btn-primary)

Well done in a few steps you have configured a new rails app to use twitter bootstrap, The actual configuration doesn't take long at all and there are only a few steps required before you are up and running and ready to use bootstrap classes for styling.

An Email Mailchimp App

For this rails app you will create an email form to submit an email however the app will connect to the mailchimp api and submit the email to a list. You will use the gibbon gem to simplify the process of connecting to the mailchimp api.

To build this app you will need to sign up for a mailchimp account. Go to **mailchimp.com** and sign up for a free account. You will need two things to build this app, a mailchimp **api key** so that you can connect to the mailchimp service, and a **list id** When you create a members list it will have an id, this is used to subscribe people to that particular list.

An api key is a line of code that gets passed between computer programs and is used to identify the person(computer) that wants to use the website. We want to use the mailchimp website so we need an api key so that we can be identified.

Once you have signed up click on the **dropdown menu** in the toolbar at the top, it will be named after your name/username. Click the **account button**.

You will now be on your account page which will show an overview for you. You will see another toolbar/menu below. Click the **extras** dropdown and then select **api keys**.

On the api key page you will see a button called **create a key** Click it and a new api key will be created. Copy it down as you will use it in your app.

Now in the menu at the top click the **lists** link, You will see a button **create list**, click that button and fill out the details on the next page. Once you are done at the bottom click the **save** button.

Click the **list** link at the top of the page again, Next to your newly created list is a **stats** dropdown menu, Click the dropdown arrow and click on **settings**, At the bottom of the settings page you will see a **unique list id**, copy it down as this is the id for the list you just created.

Also if you found the mailchimp website a little bit confusing don't worry as I do too, you soon get used to it though.

Why did you need to get an api key and a list id again? It is so that when users subscribe to your app (hypothetically as this is a test app) The email will get subscribed to the list you just created, You can then email users and keep them updated.

Mailchimp setup done!

Make sure you are in the workspace directory:

```
~/workspace $ pwd
/home/ubuntu/workspace
```

Generate a new rails app called email_mailchimp

```
An email_mailchimp app
/workspace

$ rails _4.2.5_ new email_mailchimp
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  create  app/helpers/application_helper.rb
  create  app/views/layouts/application.html.erb
  create  app/assets/images/.keep
  create  app/mailers/.keep
  ...
```

Change directory into the new app:

```
$ cd email_mailchimp/
~/workspace/email_mailchimp $
```

Run the server: Check all is fine.

```
An email_mailchimp app
/workspace/email_mailchimp

$ rails server -b $IP -p $PORT
```

Generate a User model

```
An email_mailchimp app
/workspace/email_mailchimp

$ rails generate model User email:string
Running via Spring preloader in process 788
  invoke  active_record
  create  db/migrate/20160305100909_create_users.rb
  create  app/models/user.rb
  invoke  test_unit
```

```
create test/models/user_test.rb
create test/fixtures/users.yml
```

Here in the generate command email:string is added so that the migration file automatically has the email column in it. Open the migration file to see.

Migrate the database

```
An email_mailchimp app
/workspace/email_mailchimp

$ rake db:migrate
== 20160305100909 CreateUsers: migrating =====
-- create_table(:users)
  -> 0.0019s
== 20160305100909 CreateUsers: migrated (0.0020s) =====
```

You can view the /db/schema.rb file to see the users table that got generated. It has a column named email of type string.

Generate the Users controller and actions

```
An email_mailchimp app
/workspace/email_mailchimp

$ rails generate controller Users index new
Running via Spring preloader in process 819
  create  app/controllers/users_controller.rb
  route   get 'users/new'
  route   get 'users/index'
  invoke  erb
...

```

Edit the routes.rb file

```
An email_mailchimp app
/workspace/email_mailchimp/config/routes.rb

Rails.application.routes.draw do
  resources :users, only: [:index, :new, :create]
  root 'users#index'
...

```

Add the root route and set it to the index page. If you remember the resources line it automatically creates lots of routes for our controller/actions. In this case the users controller. These are called restful routes and they are common routes that are used in web app development.

Seven different routes get created and are mapped to controller actions.

The routes are index, new, create, show, edit, update, destroy. See [rails guides](#) for more information and check out section 3.2 for a table showing you the combinations. Just looking at the table of routes, actions and http verbs doesn't do it justice, or help you learn, So the routes will be covered more later on.

Note the **only: [:index, :new, :create]** line, This tells rails that you want to create the restful routes but only the ones named there. For example we don't need the destroy action/route so don't generate it. Use the rake routes command to see the reduced amount of routes.

Run the server:

```
An email_mailchimp app
/workspace/email_mailchimp

$ rails server -b $IP -p $PORT
```

Update the Users controller:

```
An email_mailchimp app
/workspace/email_mailchimp/app/controllers/users_controller.rb

class UsersController < ApplicationController
  def index
    end

  def new
    @user = User.new
    end

  def create
    @user = User.create(user_params)
    if @user.save
      flash[:success] = "Thank you for signing up to the mailing list!"
      redirect_to root_path
    else
      flash[:error] = "There was an error, try again"
      render :new
    end
  end

  private

  def user_params
    params.require(:user).permit(:email)
  end
end
```

Add the create method and private user_params method.

Update the index and new pages:

```
An email_mailchimp app  
/workspace/email_mailchimp/app/views/users/index.html.erb
```

```
<%= link_to "SUBSCRIBE", new_user_path %>
```

```
An email_mailchimp app  
/workspace/email_mailchimp/app/views/users/new.html.erb
```

```
<%= form_for @user do |f| %>  
  <%= f.email_field :email, :placeholder => "Email" %>  
  <%= f.submit "SUBMIT" %>  
<% end %>
```

Add a link in the index page and a form in the new page.

Update application layout file to show flash messages:

```
An email_mailchimp app  
/workspace/email_mailchimp/app/views/layouts/application.html.erb
```

```
<!DOCTYPE html>  
<html>  
<head>  
  <title>EmailMailchimp</title>  
  <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>  
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>  
  <%= csrf_meta_tags %>  
</head>  
<body>  
  
  <% flash.each do |name, message| %>  
    <%= content_tag :div, message, class: name %>  
  <% end %>  
  
  <%= yield %>  
  
</body>  
</html>
```

Add basic validations to the User model file:

```
An email_mailchimp app  
/workspace/email_mailchimp/app/models/user.rb
```

```
class User < ActiveRecord::Base  
  
  validates :email, presence: true
```

```
  validates :email, length: {minimum: 5}
end
```

Run the server:

```
An email_mailchimp app
/workspace/email_mailchimp

$ rails server -b $IP -p $PORT
```

Test that you get a success flash message when you subscribe your email, and also an error flash message if your email is too short.

Add the gibbon gem to the gemfile:

```
An email_mailchimp app
/workspace/email_mailchimp/gemfile

source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.5'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.1.0'
# See https://github.com/rails/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes following links in your web application faster. Read more:
https://github.com/rails/turbolinks
gem 'turbolinks'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.0'
# bundle exec rake doc:rails generates the API under doc/api.
gem 'sdoc', '~> 0.4.0', group: :doc
gem 'gibbon', '~> 2.2', '>= 2.2.1'
...
```

Run bundle install

```
An email_mailchimp app
/workspace/email_mailchimp
```

```
$ bundle install
```

Add your mailchimp api key and list id to your secrets.yml file

```
An email_mailchimp app
/workspace/email_mailchimp/config/secrets.yml

...

development:
  secret_key_base:
bc65b2dfe8afd3dc0cbdbe1bbbee3811d0fa94088936711c356eb3dc9fc8548227eb3bc70527e56d6ca8b9e1c762
2e80ac6f117d6bdb31e79f23c8fc6222a8d3
  mailchimp_api_key: hfuksolfjd83j38d39ik9skj989dk
  list_id: h3g51l89sf

test:
  secret_key_base:
67ef21b8a50815bfa531a4fdcd4beae45634589b53f2aee3976abd6ddfe526c1193583e7e612773f419d6dce3171
fdeb176cb154735d40537fa6eedcf2e007f6

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

In your secrets.yml file add your api key and list id, use a space between the colon and the key/code. Those keys above are fake they won't work, you will need your own by signing up. Also this is not normally how you would add sensitive information to you app, you would use environment variables which I will cover later. However since this is only a test app running in development mode you can add your keys to the secrets file under development.

Add a subscribe method to subscribe the email to mailchimp:

```
An email_mailchimp app
/workspace/email_mailchimp/app/models/user.rb

class User < ActiveRecord::Base

  validates :email, presence: true
  validates :email, length: {minimum: 5}

  before_save :downcase_email
  after_save :subscribe

  def downcase_email
    self.email = email.downcase
  end

  def subscribe
    gibbon = Gibbon::Request.new(api_key: Rails.application.secrets.mailchimp_api_key)
    list_id = Rails.application.secrets.list_id
```



```

begin
  gibbon.lists(list_id).members.create(
    body: {
      email_address: self.email,
      status: "subscribed"
    })
  rescue Gibbon::MailChimpError => e
    puts "Try Again: #{e.message} - #{e.raw_body}"
  end
end
end
end

```

If you look at the user model you will see a method called `downcase_email`, this method calls the email attribute for the current user (`self`) and converts the characters to lowercase.

Look at the line before `_save :downcase_email`

This is a callback method that runs before the data is saved to the database. So Before we save the email to the database run the `downcase_email` method.

If you look further down you will see the `subscribe` method.

The method starts with a local variable called `gibbon` which creates a new gibbon instance request. It uses the api key in the `secrets.yml` file. The api key gets passed via the line:

```
api_key: Rails.application.secrets.mailchimp_api_key
```

You can see we target the `secrets` file and the `mailchimp_api_key` variable.

Then in the `list_id` variable we also store the list id from the `secrets.yml` file with the line:

```
list_id = Rails.application.secrets.list_id
```

With these two pieces of information you can now target a specific list (`list_id`) and subscribe an email to it.

Next is a `begin rescue` block that is used to catch any errors and print them to the console. In the `begin` section we have the line `gibbon.lists(list_id).members.create` which will target the list with the id stored in the `list_id` variable and create a new member, (subscribe a new email to the list).

Then you have the body of the create method, This is what gets passed to the api and is added to your mailchimp list. In the body we add the email address with `self.email` and we add a status of `subscribed`. The status is needed in order to add an email address to a mailchimp list.

Below the body we have a rescue statement, If there is an error it gets stored in the variable `e`. The error message then gets `puts` (printed to the console screen) The `e.message` prints the error message and `e.raw_body` gives you greater information about the error.

Run the server and subscribe some emails. Then log into your mailchimp account and you will see the subscribed emails when you click on the list. If you type in something that is obviously a fake email then it may not work. Check the console to see if there was an error.

WELL DONE!

You created an app that subscribes an email to a mailchimp list. You learned how to use an api to connect to another system/service. You learned a little bit about storing keys/code in the secrets.yml file and you also learned about callback methods that can be run at certain times during the apps processing (before saving to the database, after saving to the database etc.).

```

<div class="container">
  <div class="row">
    <div class="col-md-6 col-lg-8"> <!-- _____ BEGIN NAVIGATION
      <nav id="nav" role="navigation">
        <ul>
          <li><a href="index.html">Home</a></li>
          <li><a href="home-events.html">Home Events</a></li>
          <li><a href="multi-col-menu.html">Multiple Column Men
          <li class="has-children"> <a href="#" class="current">
            <ul>
              <li><a href="tall-button-header.html">Tall But
              <li><a href="image-logo.html">Image Logo</a></
              <li class="active"><a href="tall-logo.html">Ta
            </ul>
          </li>
          <li class="has-children"> <a href="#">Carousels</a>
            <ul>
              <li><a href="variable-width-slider.html">Variab
              <li><a href="variable-width-slider.html">Testimon1

```

Using figaro to store sensitive information

This app will be similar to the last however it will cover the use of the figaro gem. This gem is used for storing sensitive information in environment variables. It is important to know this as whilst it is okay to put api_keys and code in your secrets.yml file it is not ideal.

Make sure you are in the workspace directory:

```
~/workspace $ pwd
/home/ubuntu/workspace
```

Generate a new rails app called email_mailchimp

```
An email app with figaro
/workspace
$ rails _4.2.5_ new email_with_figaro
  create
  create README.rdoc
...
```

Change directory into the new app:

```
$ cd email_with_figaro/
~/workspace/email_with_figaro $
```

Run the server: Check all is fine.

```
An email app with figaro
/workspace/email_with_figaro
$ rails server -b $IP -p $PORT
```

Generate a User model

```
An email app with figaro
/workspace/email_with_figaro
$ rails generate model User email:string
Running via Spring preloader in process 788
```

Migrate the database

```
An email app with figaro  
/workspace/email_with_figaro
```

```
$ rake db:migrate  
== 20160305100909 CreateUsers: migrating =====  
-- create_table(:users)  
   -> 0.0019s  
== 20160305100909 CreateUsers: migrated (0.0020s) =====
```

Generate the Users controller and actions

```
An email app with figaro  
/workspace/email_with_figaro
```

```
$ rails generate controller Users index new  
Running via Spring preloader in process 819  
...
```

Edit the routes.rb file

```
An email app with figaro  
/workspace/email_with_figaro/config/routes.rb
```

```
Rails.application.routes.draw do  
  resources :users, only: [:index, :new, :create]  
  root 'users#index'  
  ...
```

Run the server:

```
An email app with figaro  
/workspace/email_with_figaro
```

```
$ rails server -b $IP -p $PORT
```

Update the Users controller:

```
An email app with figaro  
/workspace/email_with_figaro/app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController  
  def index
```

```

end

def new
  @user = User.new
end

def create
  @user = User.create(user_params)
  if @user.save
    flash[:success] = "Thank you!"
    redirect_to root_path
  else
    flash[:error] = "There was an error!"
    render :new
  end
end

private

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

```

Update the index and new pages:

An email app with figaro
/workspace/email_with_figaro/app/views/users/index.html.erb

```
<%= link_to "SUBSCRIBE", new_user_path %>
```

An email app with figaro
/workspace/email_with_figaro/app/views/users/new.html.erb

```

<%= form_for @user do |f| %>
  <%= f.email_field :email, :placeholder => "Email" %>
  <%= f.submit "SUBMIT" %>
<% end %>

```

Update application layout file to show flash messages:

An email app with figaro
/workspace/email_with_figaro/app/views/layouts/application.html.erb

```

<!DOCTYPE html>
<html>
<head>
  <title>EmailWithFigaro</title>
  <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>

```

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

<%= yield %>

</body>
</html>
```

Add basic validations to the User model file:

```
An email app with figaro
/workspace/email_with_figaro/app/models/user.rb

class User < ActiveRecord::Base
  validates :email, presence: true
  validates :email, length: {minimum: 5}
end
```

Run the server:

```
An email app with figaro
/workspace/email_with_figaro

$ rails server -b $IP -p $PORT
```

Test that you get a success flash message when you subscribe your email, and also an error flash message if your email is too short.

Add the gibbon gem to the gemfile and the figaro gem:

```
An email app with figaro
/workspace/email_with_figaro/gemfile

source 'https://rubygems.org'

...

# Turbolinks makes following links in your web application faster. Read more:
https://github.com/rails/turbolinks
gem 'turbolinks'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.0'
# bundle exec rake doc:rails generates the API under doc/api.
gem 'sdoc', '~> 0.4.0', group: :doc
gem 'gibbon', '~> 2.2', '=> 2.2.1'
```

```
gem "figaro", "~> 1.1.1"
...
```

Run bundle install

```
An email app with figaro
/workspace/email_with_figaro
```

```
$ bundle install
```

Run bundle exec figaro install

```
An email app with figaro
/workspace/email_with_figaro
```

```
$ bundle exec figaro install
  create  config/application.yml
  append  .gitignore
```

As you can see this figaro command creates a file called application.yml in the config folder. The append .gitignore means that if you are using version control such as git, then if you push/submit your work publicly that file will be ignored. This means you can safely add information such as your api keys as it won't be submitted anywhere (If you are using version control).

Add your mailchimp api key and list id to the application yml file (no tabs just spaces)

```
An email app with figaro
/workspace/email_with_figaro/config/application.yml
```

```
# Add configuration values here, as shown below.
#
# pusher_app_id: "2954"
# pusher_key: 7381a978f7dd7f9a1117
# pusher_secret: abdc3b896a0ffb85d373
# stripe_api_key: sk_test_2J0l093x0yw72XUYJHE4Dv2r
# stripe_publishable_key: pk_test_ro9jV5SNwGb1yYlQfzG17LHK
#
# production:
#   stripe_api_key: sk_live_EeHnL644i6zo4Iyq4v1KdV9H
#   stripe_publishable_key: pk_live_9lctxpSIHbGwmd094101XVU
MAILCHIMP_API_KEY: hfukso1fjd83j38d39ik9skj989dk
LIST_ID: h3g51l89sf
```

We will target these environment variables in our secrets.yml file.

Target the environment variables in the secrets.yml file:

```
An email app with figaro
/workspace/email_with_figaro/config/secrets.yml

...

# Make sure the secrets in this file are kept private
# if you're sharing your code publicly.

development:
  secret_key_base:
0e8108dc2f3b58b38c75734cadf6842b7472fc7ebd6191f5c37a1a30cefee176687d7d45a68666f9a3b7ec9c5093
86e9fca1bc58deca2faf35743f890791f2e1
  mailchimp_api_key: <%= ENV["MAILCHIMP_API_KEY"] %>
  list_id: <%= ENV["LIST_ID"] %>
test:
  secret_key_base:
bc8b9479f36a56748cc7c8902e5f18b37cce19abbabceb4e019c288f6e60cea6847a847b1cc9d9d183131bbf34da
74a9bae0397ac8c010ae5f926b33ef2d5c31

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

We are now targeting the the variables in our application.yml file using the ENV environment variable and the name of the variable in the application.yml file. As you can see we don't have the api_key in this file so the application is now safer especially if submitting the code publicly.

Add a subscribe method to subscribe the email to mailchimp:

```
An email app with figaro
/workspace/email_with_figaro/app/models/user.rb

class User < ActiveRecord::Base

  validates :email, presence: true
  validates :email, length: {minimum: 5}

  after_save :subscribe

  def subscribe
    gibbon = Gibbon::Request.new(api_key: Rails.application.secrets.mailchimp_api_key)
    list_id = Rails.application.secrets.list_id

    begin
```



```
gibbon.lists(list_id).members.create(  
  body: {  
    email_address: self.email,  
    status: "subscribed"  
  })  
rescue Gibbon::MailChimpError => e  
  puts "Try Again: #{e.message} - #{e.raw_body}"  
end  
end  
end
```

As you can see with the gibbon local variable we are still targeting the mailchimp_api_key in the secrets.yml file,

```
gibbon = Gibbon::Request.new(api_key: Rails.application.secrets.mailchimp_api_key)
```

But if you look in that file you will see environment variables are now being used to store the information.

WELL DONE!

Cool you recreated the app using the figaro gem and environment variables. You may think this was a bit of a pointless app as nothing much was new however it is important to know how to use and store information that is more sensitive. You don't want other people to access your api keys or any other information. You can now build an app knowing your information is safe.

An explanation on environment variables:

Environment variables are used to keep sensitive information secure.

Let's say you were connecting to the Gmail api. You could use environment variables for the username and password.

```
user_name: ENV["GMAIL_USERNAME"]
```

```
password: ENV["GMAIL_PASS"]
```

ENV tells rails to look for an environment variable with the name "GMAIL_USERNAME" or "GMAIL_PASS" on the server that the app is running on.

So in our earlier example we were running on the local server, and in the secrets.yml file you can see we placed the variables under the development line of code. (We are building the app). If we were going to use the app in real life then we would add ENV variables beneath the production line of code and push our code to a live server. The ENV variables are divided up depending on the environment in which the app is running.

There are various ways to set ENV variables for use, a yml file is in my opinion one of the easier approaches, especially when a gem like figaro can be used to help generate the files for you and add the file to the .gitignore file automatically. The gem does the heavy lifting and you don't need to reinvent the wheel.

Make sure you are in your previous app cd into email_with_figaro:

```
An email app with figaro
/workspace/email_with_figaro

$ rails console
Running via Spring preloader in process 1038
Loading development environment (Rails 4.2.5)
2.3.0 :001 > ENV['MAILCHIMP_API_KEY']
=> "hfuksolfjd83j38d39ik9skj989dk"
2.3.0 :002 >
2.3.0 :002 > exit
~/workspace/email_with_figaro $
```

The rails console command lets you interact with your app from the command line.

You can use it to make quick checks when developing your app etc.

Type the rails console command to open up the console.

Then type the ENV['MAILCHIMP_API_KEY'] ENV variable into the console. The variable will be printed out.

Look above and you will see the line says loading development environment, If you had different ENV variables for production and a live server then they would get loaded not the development ENV variables. However since the console is running in development mode these are the variables that get loaded.

A Posts app

In this app you will utilize all of the restful routes/actions index, new, create, show, edit, update, destroy, and build an app that allows you to create a new post with a title and body. You will then be able to edit the post and update its contents as well as delete it.

Make sure you are in the workspace directory:

```
~/workspace $ pwd
/home/ubuntu/workspace
```

Generate a new rails app called posts

```
A Posts app
/workspace
$ rails _4.2.5_ new posts
  create
  create README.rdoc
...
```

Change directory into the new app:

```
$ cd posts/
~/workspace/posts $
```

Run the server: Check all is fine.

```
A Posts app
/workspace/posts
$ rails server -b $IP -p $PORT
```

Generate a Post model

```
A Posts app
/workspace/posts
```

```
$ rails generate model Post title:string body:text
Running via Spring preloader in process 1147
```

Migrate the database

```
A Posts app
/workspace/posts

$ rake db:migrate
== 20160306094443 CreatePosts: migrating =====
-- create_table(:posts)
   -> 0.0012s
== 20160306094443 CreatePosts: migrated (0.0013s) =====
```

check out your schema.rb file to see the posts table that got generated.

Generate the Posts controller and actions

```
A Posts app
/workspace/posts

$ rails generate controller Posts index new edit show
Running via Spring preloader in process 819
...
```

There is nothing new here, the output to the console is larger as we are creating more actions and views with the generate command. If you look at what got created you will see all the associated view files.

Edit the routes.rb file

```
A Posts app
/workspace/posts/config/routes.rb

Rails.application.routes.draw do
  resources :posts
  root 'posts#index'
  ...
```

This time we will be wanting all the resources for the posts controller.

Run the server: Try going to the various pages /posts/new

```
A Posts app  
/workspace/posts
```

```
$ rails server -b $IP -p $PORT
```

Update the posts controller:

```
A Posts app  
/workspace/posts/app/controllers/posts_controller.rb
```

```
class PostsController < ApplicationController  
  def index  
    @posts = Post.all  
  end  
  
  def new  
    @post = Post.new  
  end  
  
  def create  
    @post = Post.create(post_params)  
    if @post.save  
      flash[:success] = "You created a new post"  
      redirect_to @post  
    else  
      flash[:danger] = "There was an error."  
      render :new  
    end  
  end  
  
  def edit  
    @post = Post.find(params[:id])  
  end  
  
  def update  
    @post = Post.find(params[:id])  
    if @post.update(post_params)  
      flash[:success] = "Post updated."  
      redirect_to @post  
    else  
      flash[:danger] = "There was an error when updating."  
      render :edit  
    end  
  end  
  
  def show  
    @post = Post.find(params[:id])  
  end  
  
  def destroy  
    @post = Post.find(params[:id])  
    @post.destroy  
  
    redirect_to root_path  
  end  
end
```

```
private
  def post_params
    params.require(:post).permit(:title, :body)
  end
end
```

There are quite a few new actions going on there but it is quite similar to what you have been doing. Take a look at the show action. The code in that action uses the params hash and looks for an id, it then displays the post with that id.

```
@post = Post.find(params[:id])
```

The id is passed from the url, each new post that gets created in the table will have an id attribute, you won't see this column on the table in the schema.rb file but it is generated automatically for every table and it is there. The first post you create will have an id of 1 and the second an id of 2 and so on. That is why the line above looks for the id parameter in the params hash (which comes from the url) which it then uses to display a particular post.

If you do the **rake routes** command you will see this line

```
post GET    /posts/:id(.:format)    posts#show
```

You can see the /posts/:id URI pattern, This shows us that we need a unique id in order for this route to work. Remember the id is an automatically generated column in your table.

The post that gets found with the find method and id passed to it from the url, is what gets stored in the @post instance variable. This instance variable will be used in the post.html.erb view later on.

As you can see by looking at the various actions we use find to find a post with a particular id quite often.

The update action is very similar to the create action, if you look we are updating which automatically saves to the database again, and we are only updating with the post_params private method. If the update was a success we redirect to that @post that just got created. (The show action) otherwise we render the edit action again.

With the destroy action we again locate a Post by id, we then call destroy on that post to delete it and finally redirect to the root_path.

Update the index page:

A Posts app

```
/workspace/posts/app/views/posts/index.html.erb
```

```
<% @posts.each do |p| %>
  <p class="title"><%= p.title %></p>
  <p class="body"><%= p.body %></p>
  <%= link_to "Edit Post", edit_post_path(p) %>
  <%= link_to "Show Post", post_path(p) %>
<% end %>
```

You cycle through each post that is stored in the `@posts` instance variable by using the `each` method. You then display the information that gets output from the erb tags. Look at the links that are being generated, run the `rake routes` command, see the `edit` and `show` routes require an `id` (URI pattern column) We pass that `id` in the `link_to` method by passing the prefix (`post_path`) the local variable which contains the `id` for that particular post. We can't use the `@posts` instance variable because that contains all the posts.

Show a particular posts information on the show page:

```
A Posts app
/workspace/posts/app/views/posts/show.html.erb
```

```
<p class="title"><%= @post.title %></p>
<p class="body"><%= @post.body %></p>
```

Using the post that is stored in the `@post` instance variable you output its title and body attributes on the show page.

Update the new.html.erb file with a form

```
A Posts app
/workspace/posts/app/views/posts/new.html.erb
```

```
<%= form_for @post do |f| %>
  <%= f.text_field :title, :placeholder => "Title" %>
  <%= f.text_field :body, :placeholder => "Body" %>
  <%= f.submit "Create Post" %>
<% end %>
```

Update the edit.html.erb file with a form

```
A Posts app
/workspace/posts/app/views/posts/new.html.erb
```

```
<%= form_for @post do |f| %>
  <%= f.text_field :title, :placeholder => "Title" %>
  <%= f.text_field :body, :placeholder => "Body" %>
  <%= f.submit "Update" %>
<% end %>
```

This form is exactly the same as the one on the new page, I just changed the button name. Also if you want to learn how to reduce the repetition of the code with these forms then you can render a partial. I won't cover it here but it allows you to render common code and store it in a single file. Not across multiple files like I have done here.

Update application layout file to show flash messages and add links:

```
A Posts app
/workspace/posts/app/views/layouts/application.html.erb

<!DOCTYPE html>
<html>
<head>
  <title>Posts</title>
  <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>
  <%= link_to "Home", root_path %>
  <%= link_to "Create Posts", new_post_path %>

  <% flash.each do |name, message| %>
    <%= content_tag :div, message, class: "alert alert-#{name}" %>
  <% end %>

  <%= yield %>

</body>
</html>
```

Add basic validations to the Post model file:

```
A Posts app
/workspace/posts/app/models/post.rb

class Post < ActiveRecord::Base
  validates :title, presence: true, length: {minimum: 3}
  validates :body, presence: true, length: {minimum: 5}
end
```

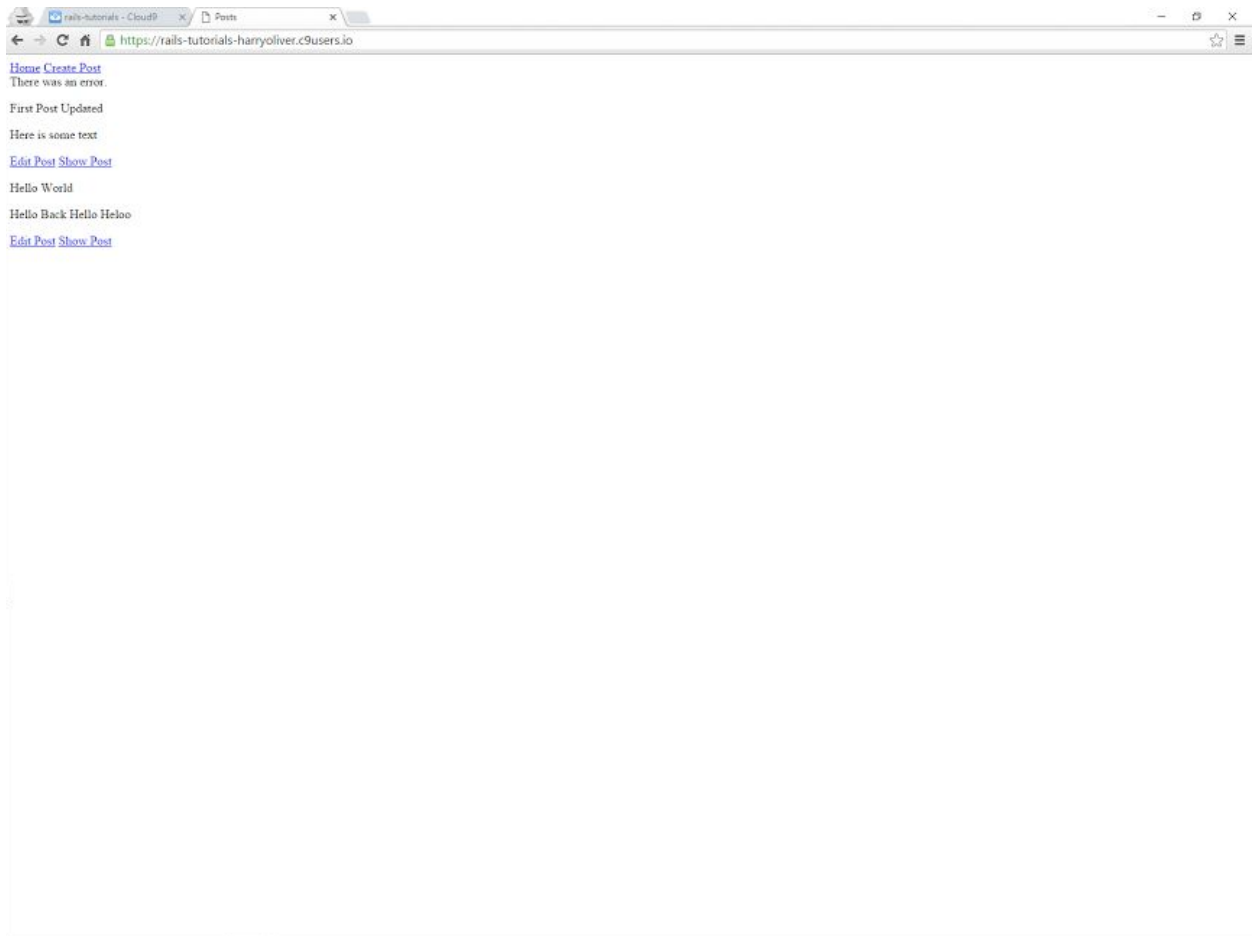
Run the server: And check flash messages display correctly, add a post and then update it, try updating information that is below the minimum length.

```
A Posts app
/workspace/posts
```



```
$ rails server -b $IP -p $PORT
```

You should now be able to add a post and edit it and update it. You should also be able to view an individual post.



Add the destroy link to the show page:

Add a link on the show page so that when you click to view a post you can then delete that post if you wish.

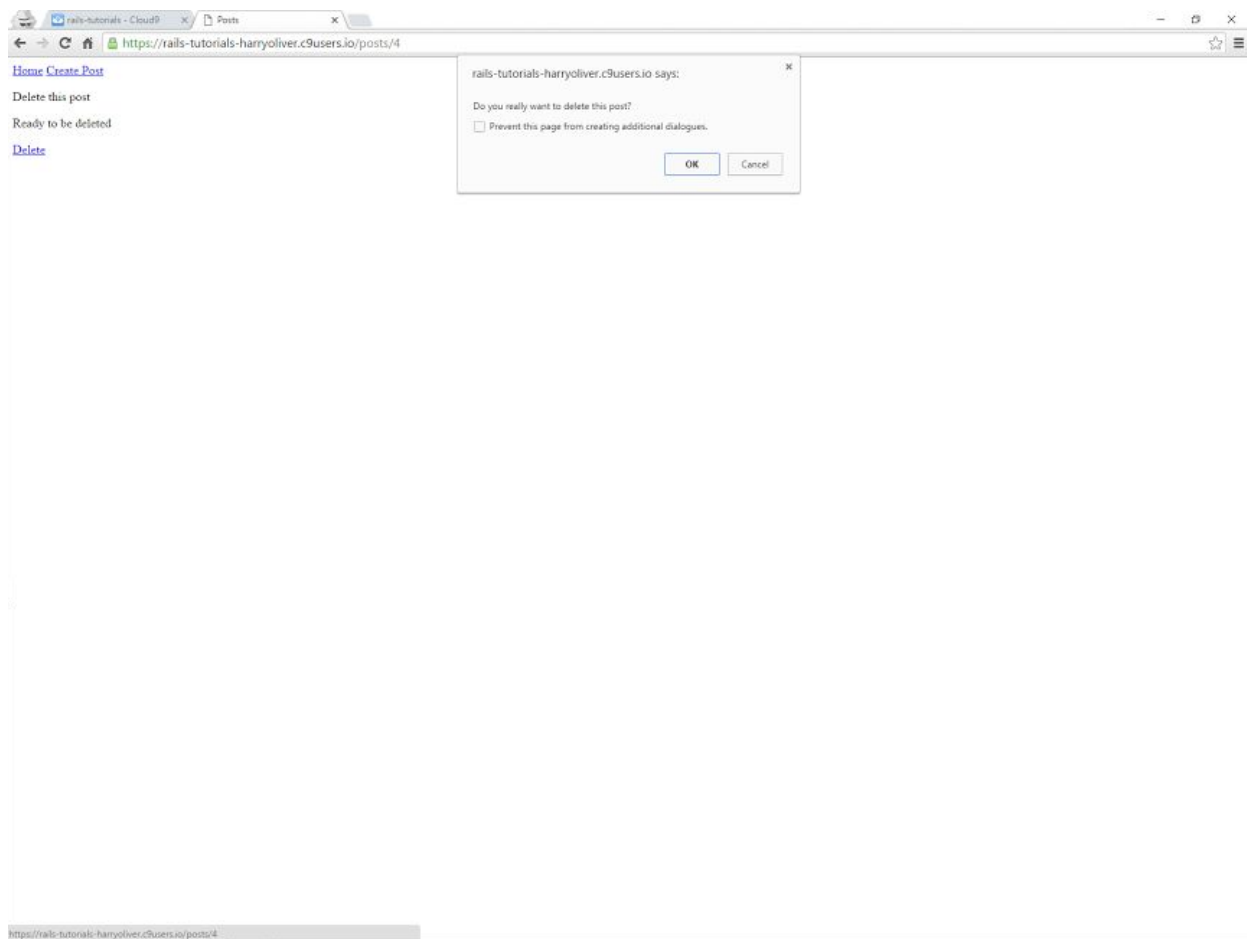
```
A Posts app  
/workspace/posts/app/views/posts/show.html.erb
```

```
<p class="title"><%= @post.title %></p>  
<p class="body"><%= @post.body %></p>  
<p><%= link_to 'Delete', post_path(@post), method: :delete, data: { confirm: "Do you really  
want to delete this post?" } %></p>
```

Here we add a `link_to` method with the text `Delete`. If you rake routes you will see the prefix for the destroy action is `post`, `_path` gets appended to this and we pass the ID of the post we want to delete in parenthesis. The `@post` instance variable contains the post as it was found using the params hash.

The `:delete` method used is then explicitly stated as an option that we add to the `link_to` method. The `data` option is also added, this provides a pop up confirmation box before the action is carried through. If you cancel then the post is not deleted.

In your javascript file you have the line `jquery_ujs` automatically included when the app was made. This is the javascript that is used to show the dialog box.



Add bootstrap styles to the app:

```
A Posts app  
/workspace/posts/app/gemfile
```

```
gem 'bootstrap-sass', '~> 3.3.6'
```

```
...
```

Add the bootstrap gem to your gemfile.

Bundle install:

```
A Posts app  
/workspace/posts/
```

```
$ bundle install
```

Rename the stylesheet with a .scss extension:

```
A Posts app  
/workspace/posts/
```

```
$ mv app/assets/stylesheet/application.css app/assets/stylesheet/application.scss
```

Update the stylesheet:

```
A Posts app  
/workspace/posts/assets/stylesheet/application.scss
```

```
...  
* file per style scope.  
*  
*/  
@import "bootstrap-sprockets";  
@import "bootstrap";  
@import "posts"; //import the posts.scss stylesheet
```

Remove the require tree lines and add the import lines.

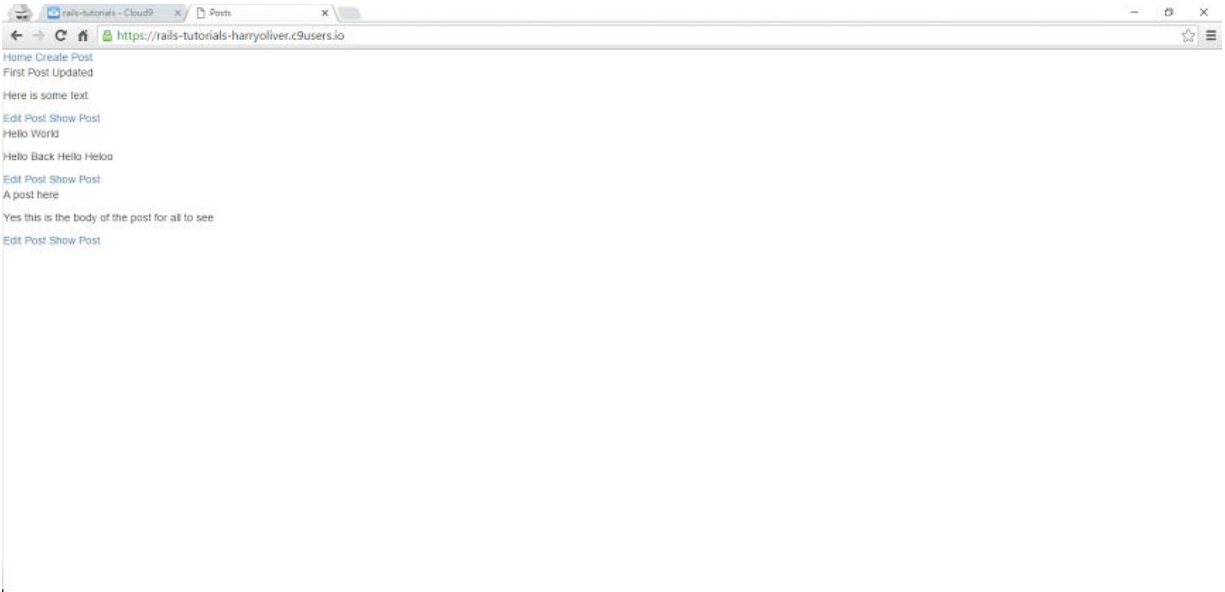
Update the javascript file:

```
A Posts app  
/workspace/posts/assets/javascripts/application.js
```

```
//  
//= require jquery  
//= require jquery_ujs  
//= require bootstrap-sprockets  
//= require turbolinks  
//= require_tree .
```

Add the bootstrap-sprockets line to the application.js file.

Start the server up and your page should be styled with some bootstrap styles:



Update application layout file with bootstrap styles:

```
A Posts app
/workspace/posts/app/views/layouts/application.html.erb

<body>

<div class="container">
  <%= link_to "Home", root_path, class: "btn btn-primary" %>
  <%= link_to "Create Post", new_post_path, class: "btn btn-success" %>

  <% flash.each do |name, message| %>
    <%= content_tag :div, message, class: "alert alert-#{name}" %>
  <% end %>

  <%= yield %>
</div>

</body>
```

Update the posts.scss file with css rules:

```
A Posts app
/workspace/posts/app/assets/stylesheets/posts.scss

.title {
```

```

font-weight: bold;
margin: .75em 0;
font-size: 1.8em;
}

.body {
background: #eee;
padding: .5em;
}

```

Add some styles to the title and body rules.

Update the edit and new pages with bootstrap styles:

```

A Posts app
/workspace/posts/app/views/posts/new.html.erb

```

```

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <div class="well">

      <p class="text-center lead">New Post</p>

      <%= form_for @post do |f| %>

        <div class="form-group">
          <%= f.text_field :title, :placeholder => "Title", class: "form-control" %>
        </div>

        <div class="form-group">
          <%= f.text_field :body, :placeholder => "Body", class: "form-control" %>
        </div>

        <%= f.submit "Create Post", class: "btn btn-primary" %>

      <% end %>

    </div>
  </div>
</div>

```

```

A Posts app
/workspace/posts/app/views/posts/edit.html.erb

```

```

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <div class="well">

      <p class="text-center lead">New Post</p>

      <%= form_for @post do |f| %>

        <div class="form-group">

```

```
<%= f.text_field :title, :placeholder => "Title", class: "form-control" %>
</div>

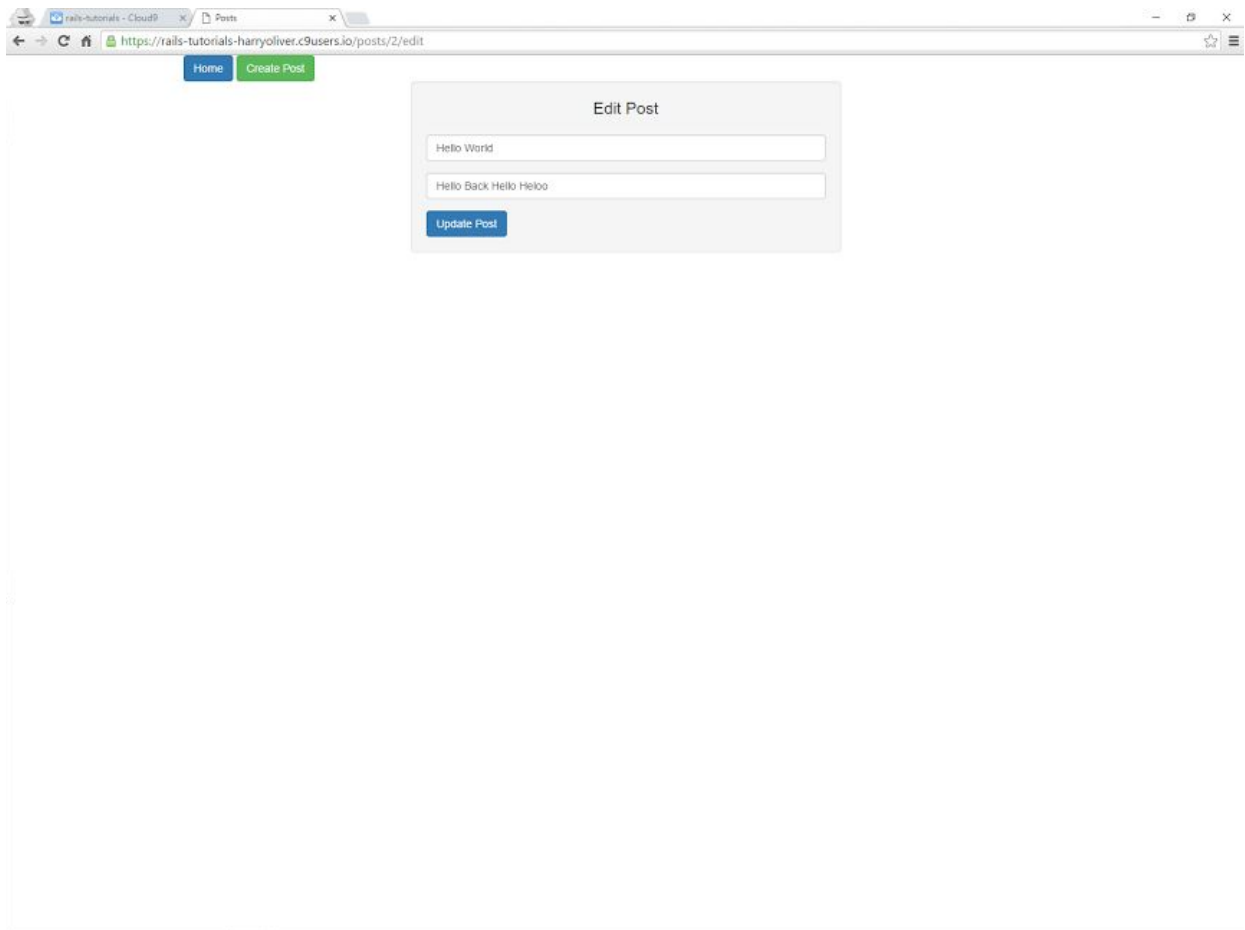
<div class="form-group">
  <%= f.text_field :body, :placeholder => "Body", class: "form-control" %>
</div>

<%= f.submit "Update Post", class: "btn btn-primary" %>

<% end %>

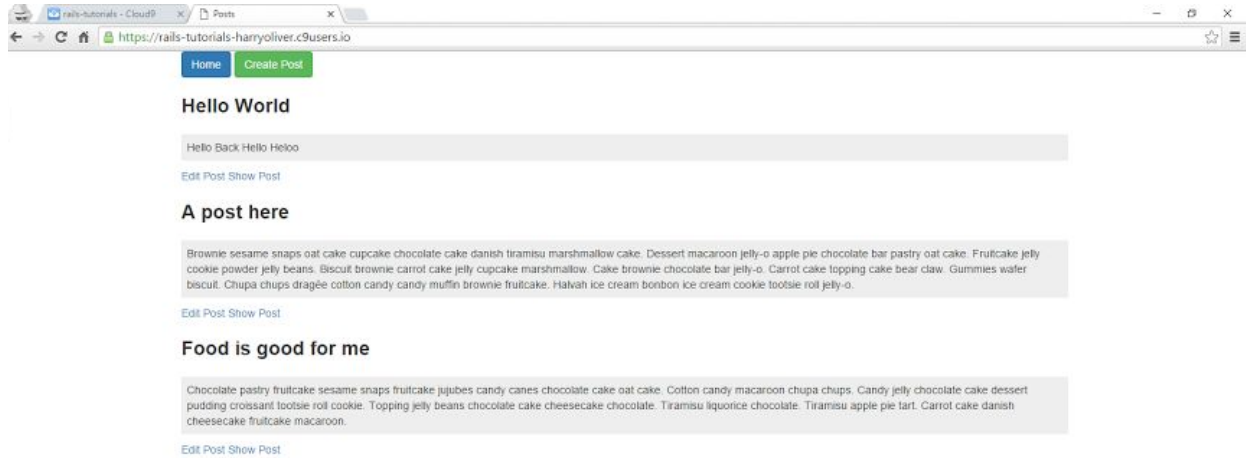
</div>
</div>
</div>
```

If you run your server and go to the new or edit page you will see a nicely styled form:



The final app should look something like this:

You should be able to add a post, show/view it, update it and delete it if needed.



WELL DONE!

You created an app that you can add a post to and view it. You can edit and update the post and even delete it. There was quite a lot to this app including installing and configuring bootstrap. Hopefully you can see how this app could be turned into a blog with a little bit more elbow grease.



FINAL THOUGHTS

You have come a long way from now knowing any ruby on rails, you have built multiple apps and learned the basics of creating your own apps. You can generate a new app, add controllers models and views to your application and even install and configure gems. You also learned a little about ENV variables and keeping sensitive data secure and also about interacting with the mailchimp api. Nicely Done.

If you enjoyed the book then could you please leave a review on amazon it would be greatly appreciated.

If you find any errors in the tutorials just drop me an email and I will correct them:
allforcoding@gmail.com

There is plenty that I haven't covered in this book such as relationships between models, rendering partials, basic authentication and much more. At the bare minimum I hope you enjoyed the book and found it useful,

Kind Regards
Harry