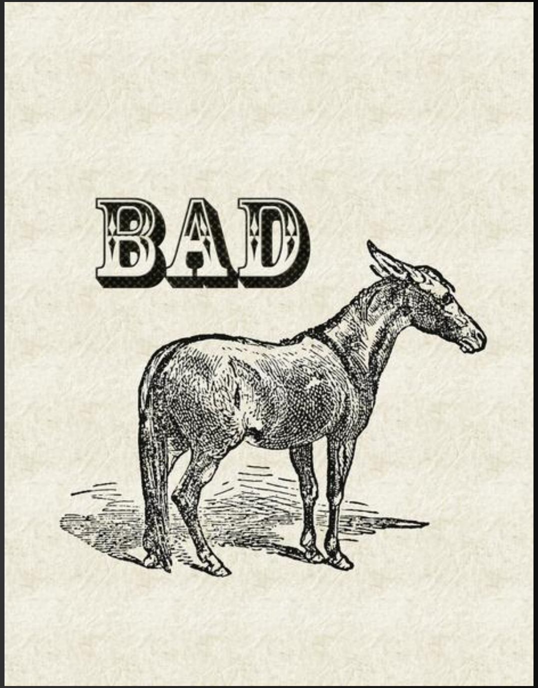
VOLKSWAGEN
FEATURE
PHILADELPHIA FLYERS

Rails!



Create a new rails app

```
gem install rails
rails new myApp -d postgresql
cd myApp
```

Open Gemfile.rb in your editor and add the following to line 2

```
ruby “2.2.2”
```

Now run:

```
bundle
```

Initialize a git repo

```
git init
git add -A
git commit -m ‘initial commit’
```

Create repo on github

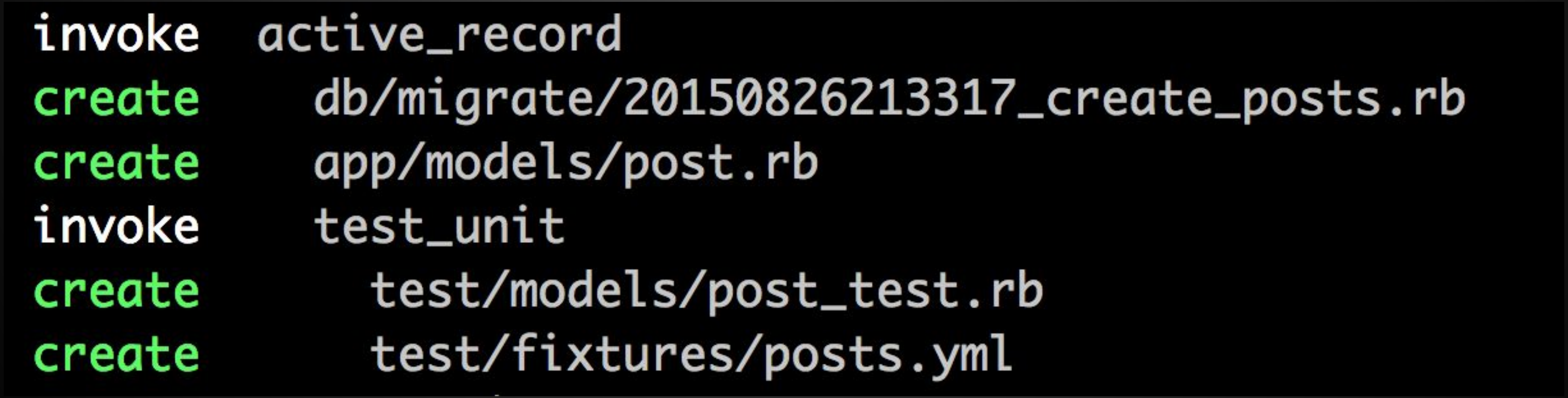
```
git add remote origin git@github.com:username/appname.git
git push origin master
```

Creating a model and rails generate

```
rails g model Post title:string author:string body:text
```

This will generate a model called Post with a title that is a string an author that is a string and a body that is text

This will create a few files for us:



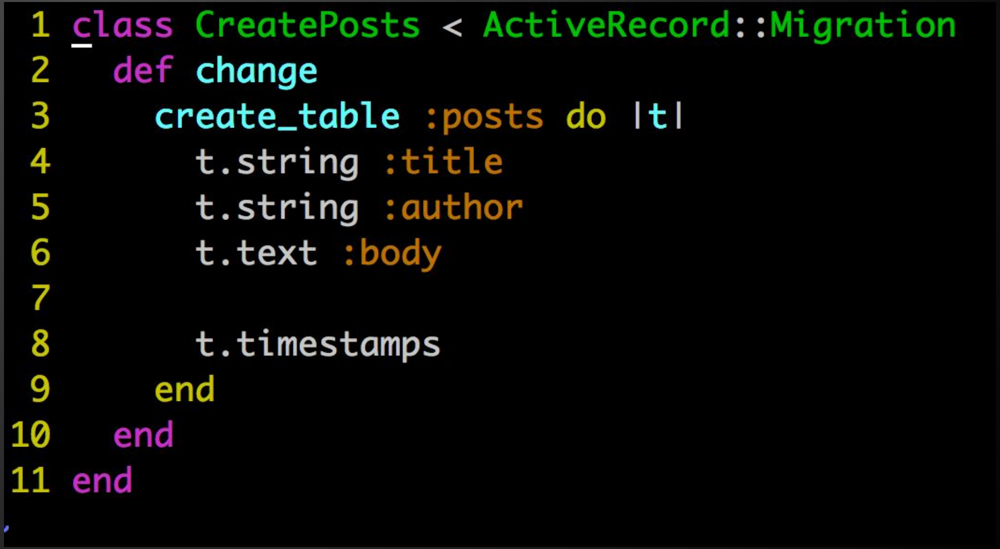
```
invoke  active_record
create    db/migrate/20150826213317_create_posts.rb
create    app/models/post.rb
invoke    test_unit
create      test/models/post_test.rb
create      test/fixtures/posts.yml
```

Model

When we generated the model a migration was created as well as a model

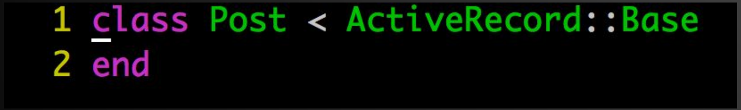
The migration tells the database what to do and the model is a class for the Post database records

db/migrate/huge_timestamp_create_posts.rb



```ruby
class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :title
      t.string :author
      t.text :body

      t.timestamps
    end
  end
end
```

app/models/post.rb



```ruby
class Post < ActiveRecord::Base
end
```

Notice the class is singular but the database records are pluralized

db:create db:migrate (rake)

Now that we created our database migration we need to run it so that our app knows about it.

Since we have not created a database we need to do that first

```
bundle exec rake db:create
```

Now if we migrate all migrations that are new will be executed

```
bundle exec rake db:migrate
```

New terms

Before moving on it's important to understand some new terms we just saw:

- bundle exec

  => Executes a command in the context of the current app bundle
- rake

  => Allows ruby to define tasks that can be run in the command line
- db:create

  => Creates the initial database (this should only be run when a project is new or you delete your database)
- db:migrate

  => Runs all new migrations that the database does not know about
- bundle exec rake -T

  => Lists all rake tasks available to the app

Rails console

The rails console is very handy and runs similar to IRB except that it is tied to you app and your apps database

There are 2 ways to enter the console:

```
bundle exec rails c
```

Or

```
bundle exec rails console
```

Inside of the console you can create a new Post from the model created earlier

```
Post.create(title: ‘My first Post’, body: ‘This is really cool’, author: ‘Me!’)
```

Hitting ctrl-c or typing `exit` will take you out of the console

Controller

Now it’s time to create a controller for posts

```
rails g controller Posts index show new
```

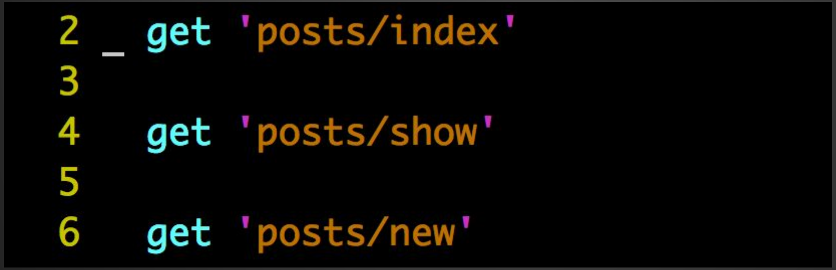
Notice that in this case Posts is pluralized

This will generate a few files for us but most importantly it will generate a posts controller. By adding index, show & new we are telling the generator that we want an index show and new methods to be in our controller. We could also add this after the fact directly into our controller but this will also generate corresponding views.

Routes

Open up config/routes.rb

Remove these lines:



```
_ get 'posts/index'

  get 'posts/show'

  get 'posts/new'
```

In their place add resources :posts

This will automatically include all of our basic crud routes for posts (PRO TIP: `rake routes` will show all routes)

Updating the controller

app/controllers/posts_controller.rb

```
def index
  @posts = Post.all
end
```

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

```
def new
  @posts = Post.new
end
```

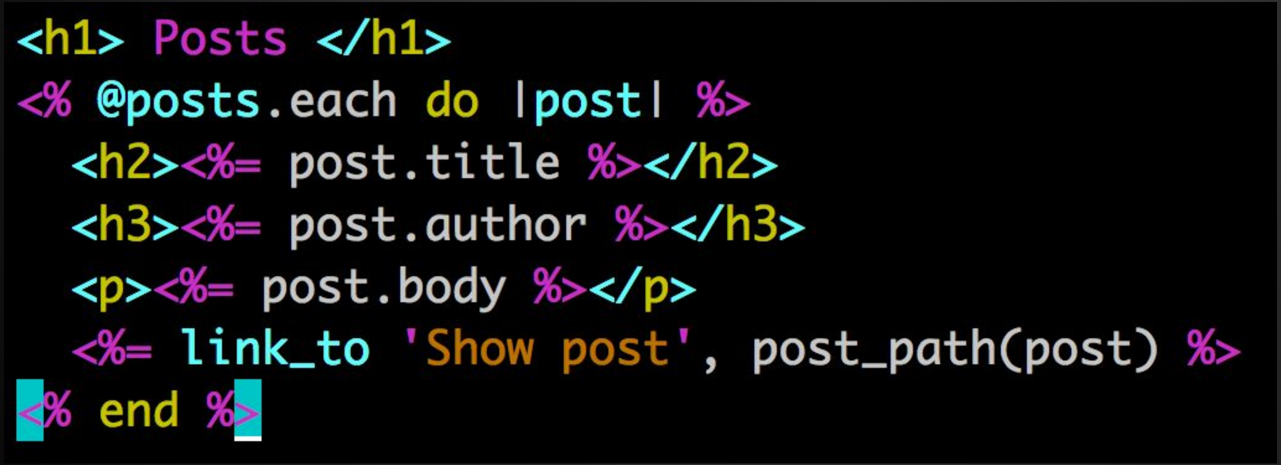
```
Post.all      => Returns all posts in the database
Post.find(#)  => Finds a post by the id passed in
Post.new      => Creates a new post in memory that is not yet saved to the db
```

Each action corresponds to a view and passes the instance variable to the view

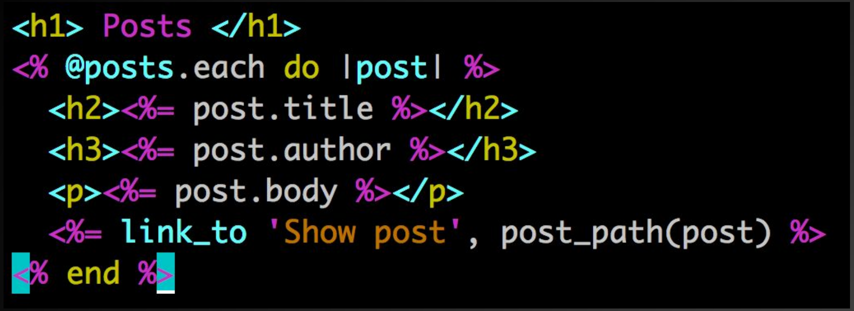
Views

Modify the post index view

app/views/posts/index.html.erb



```
<h1> Posts </h1>
<% @posts.each do |post| %>
  <h2><%= post.title %></h2>
  <h3><%= post.author %></h3>
  <p><%= post.body %></p>
  <%= link_to 'Show post', post_path(post) %>
<% end %>
```


```
<h1> Posts </h1>
<% @posts.each do |post| %>
  <h2><%= post.title %></h2>
  <h3><%= post.author %></h3>
  <p><%= post.body %></p>
  <%= link_to 'Show post', post_path(post) %>
<% end %>
```

<a href='/posts/1'>Show post</a>

@posts.each do |post| => Loops over all the posts using a post variable for each post

<h2><%= post.title %></h2> => Outputs the posts title as an h2

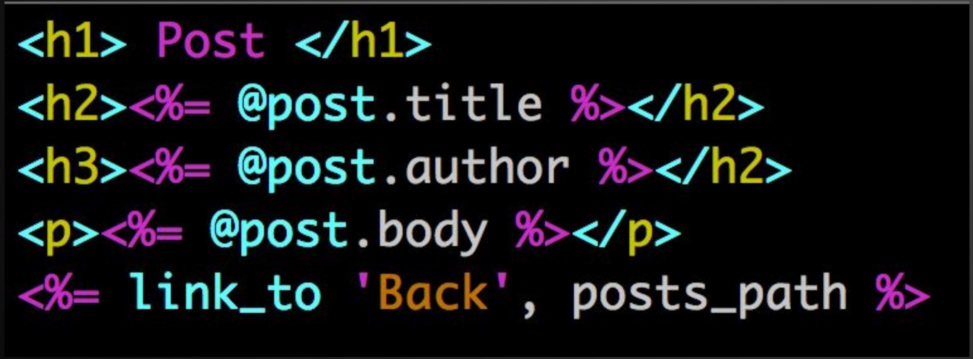
<%= link_to 'Show post', post_path(post) %>

link_to is a Rails helper method that takes 2 arguments. The first is the text to display in the link and the second is the route. In rails we can call rake routes to see the helpers for routes and then just append _path. By passing in post rails will create an id parameter of the posts id

Show view

Modify the post show view (This is what will be shown when we go to an individual post)

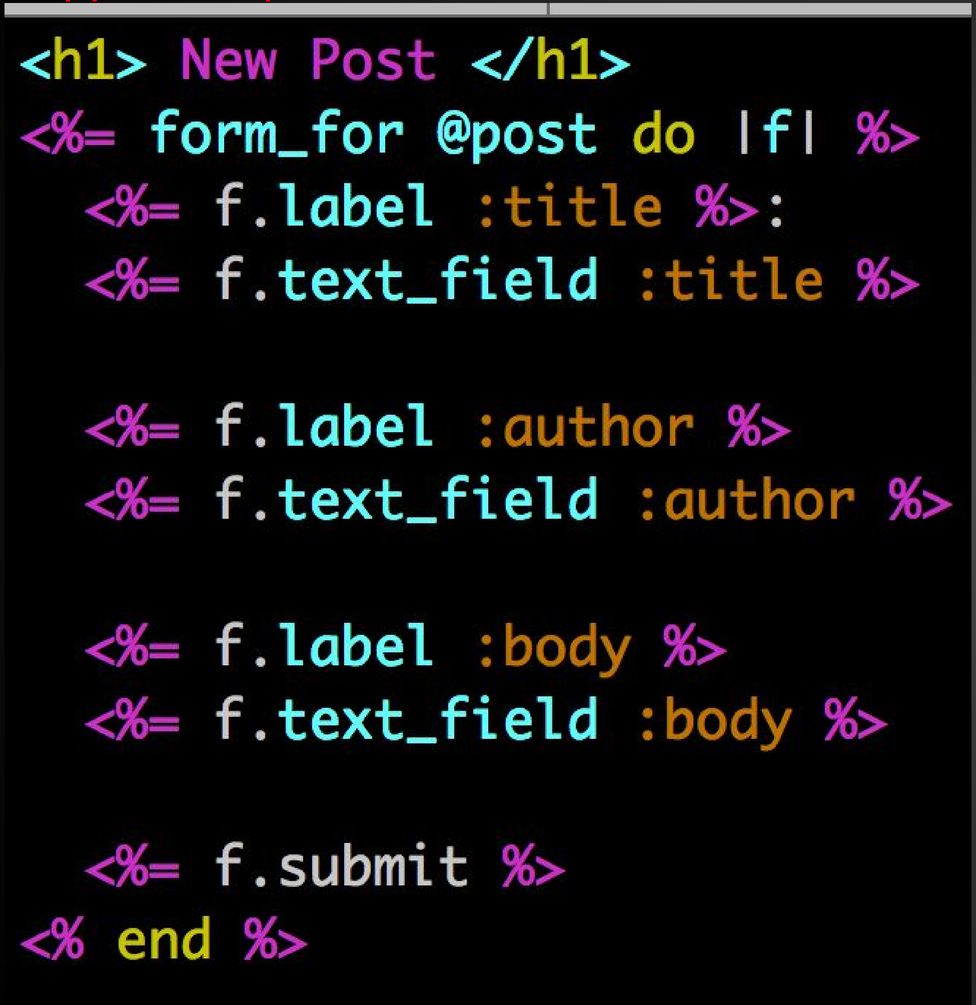
app/views/posts/show.html.erb



```
<h1> Post </h1>
<h2><%= @post.title %></h2>
<h3><%= @post.author %></h2>
<p><%= @post.body %></p>
<%= link_to 'Back', posts_path %>
```

@post comes directly from the show method in the post controller. Rails knows what to do because Rails uses convention over configuration.

app/views/posts/new.html.erb



```
<h1> New Post </h1>
<%= form_for @post do |f| %>
  <%= f.label :title %>:
  <%= f.text_field :title %>

  <%= f.label :author %>
  <%= f.text_field :author %>

  <%= f.label :body %>
  <%= f.text_field :body %>

  <%= f.submit %>
<% end %>
```

form_for @post do |f| is a rails helper that creates a form.

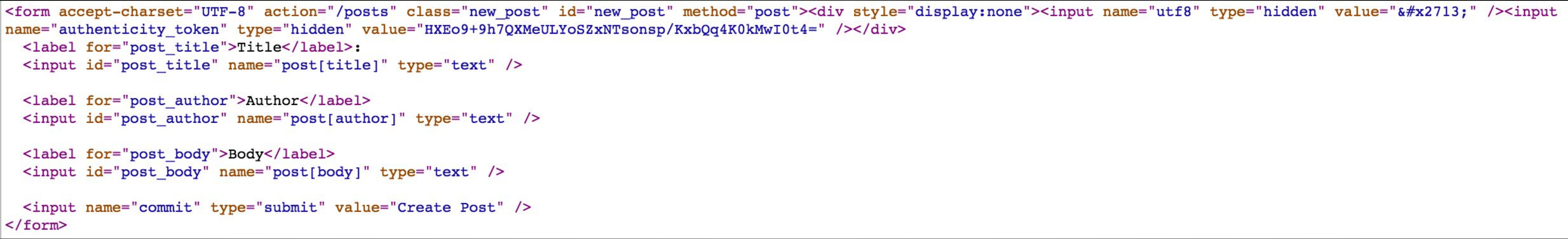
f.label creates a label

f.text_field creates an input type text

f.submit creates a submit button

form_for

Look at the HTML generated by form_for



```
<form accept-charset="UTF-8" action="/posts" class="new_post" id="new_post" method="post"><div style="display:none"><input name="utf8" type="hidden" value="&#x2713;" /><input
name="authenticity_token" type="hidden" value="HXEo9+9h7QXMeULYoSZxNTsonsp/KxbQq4K0kMwI0t4=" /></div>
  <label for="post_title">Title</label>:
  <input id="post_title" name="post[title]" type="text" />

  <label for="post_author">Author</label>
  <input id="post_author" name="post[author]" type="text" />

  <label for="post_body">Body</label>
  <input id="post_body" name="post[body]" type="text" />

  <input name="commit" type="submit" value="Create Post" />
</form>
```

The action says to route to /posts

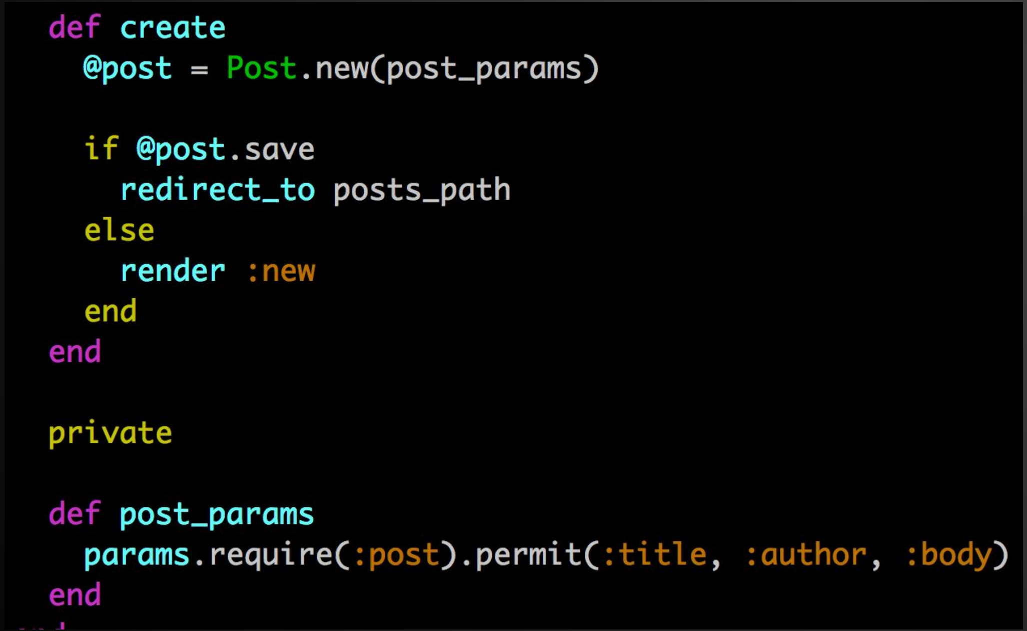
A hidden input with name authenticty_token is injected

Form inputs are all given names

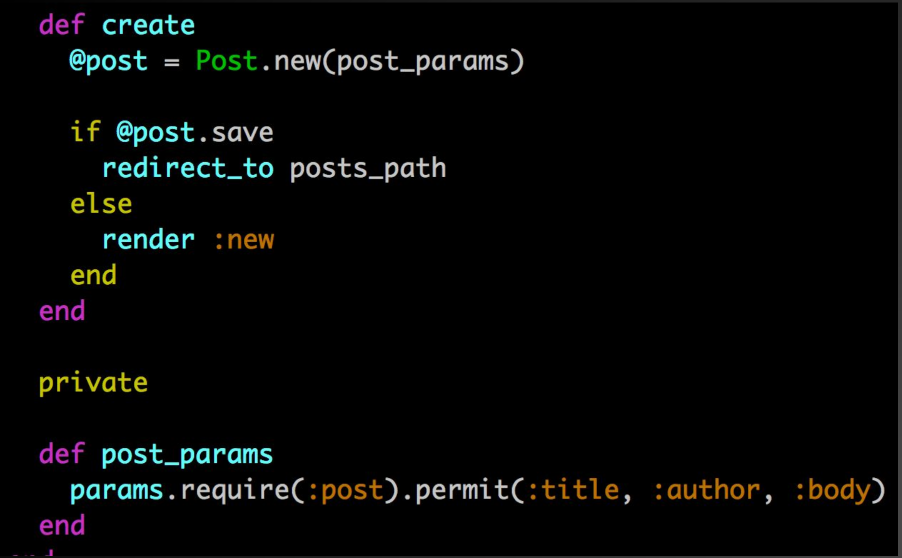
Create

Now that there is a form that creates a post the controller needs to be updated

app/controllers/posts_controller.rb



```
  def create
    @post = Post.new(post_params)

    if @post.save
      redirect_to posts_path
    else
      render :new
    end
  end

  private

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



```
def create
  @post = Post.new(post_params)

  if @post.save
    redirect_to posts_path
  else
    render :new
  end
end

private

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

- Creates a new post and passes in a method
- If the post is created in the database with the save call we redirect back to /posts
- Otherwise we redirect to the new form

- Any methods listed under private will be considered ‘private’ to be used only by this controller
- params.require(:post).permit(:title, …) is the method passed in to Post.new. This allows us to declare which parameters will be accepted for creating the post (attr_accessor). This is a security feature that prevents SQL injection

Linking

Create a link to the new post form from the index page

app/views/posts/index.html.erb

```erb
<h1> Posts </h1>
<p><%= link_to 'New Post', new_post_path %></p>
<% @posts.each do |post| %>
  <h2><%= post.title %></h2>
  <h3><%= post.author %></h3>
  <p><%= post.body %></p>
  <%= link_to 'Show post', post_path(post) %>
<% end %>
```

Rails server

To serve up a rails app simply start the server

```
bundle exec rails s
```

This starts the server on port 3000 you can also specify a port with -p

```
bundle exec rails s -p 3001
```

In a browser go to localhost:3000

Here we see the default rails root page since we have not specified one

For now just add the route to the browser URL

localhost:3000/posts

Ra

Clic
loca

Cre post is there

Clic
loca

Th PUSH

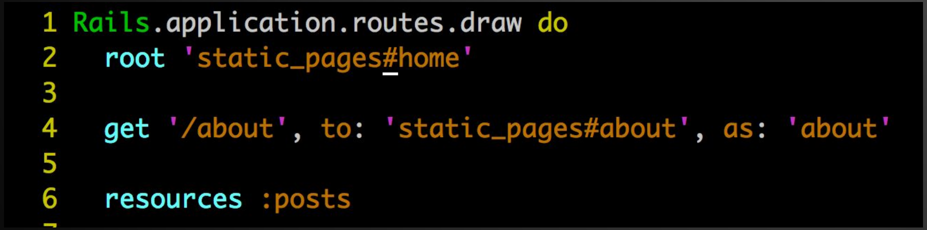
Static Pages

Some pages do not need to be dynamic like a “Home” or an “About” page

```
rails g controller StaticPages home about
```

And lets clean up the routes.rb file

config/routes.rb



```ruby
Rails.application.routes.draw do
  root 'static_pages#home'

  get '/about', to: 'static_pages#about', as: 'about'

  resources :posts
```

root tells the app where to go when you first land on the site

static_pages is the controller

#home is the method

/static_pages/about is an ugly url

so we can specify a new url

/about and point it to a controller and method and even name the helper with as: ‘about’

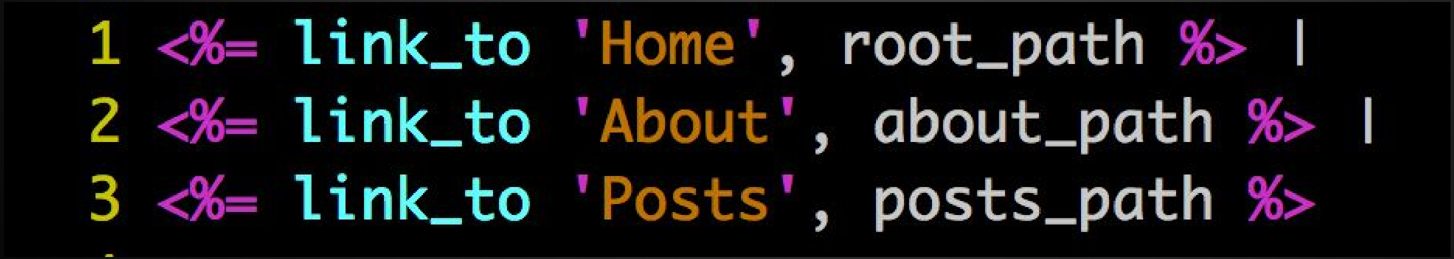
Shared views

Often there will be components that can be shared by other views such as a navbar.

Create a file in app/views/layouts called _navbar.html.erb

This is a partial meaning that it’s not an entire page just part of a page and by convention partials are named with an _ at the beginning

app/views/layouts/_navbar.html.erb



```erb
<%= link_to 'Home', root_path %> |
<%= link_to 'About', about_path %> |
<%= link_to 'Posts', posts_path %>
```

Rendering a partial app wide

Modify app/views/layouts/application.html.erb (this is the main html file shared by the entire app)

app/views/layouts/application.html.erb

```
<body>
<%= render 'layouts/navbar' %>
<%= yield %>

</body>
```

render ‘layouts/navbar tells the app to render a partial called navbar located in the layouts directory

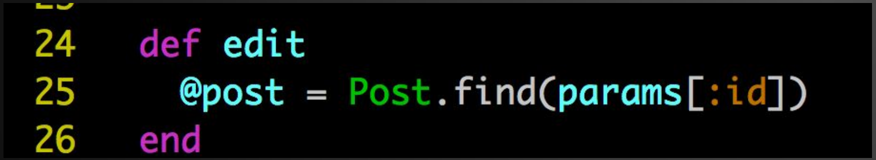
Notice there is no need for the _ or the .html.erb extension

yield tells our app to render the html from the route we are on since the navar is rendered before the yield it will appear at the top of every page

Editing a post

Update the posts controller to have an edit method & create a new edit view

app/controller/posts_controller.rb



```ruby
  def edit
    @post = Post.find(params[:id])
  end
```

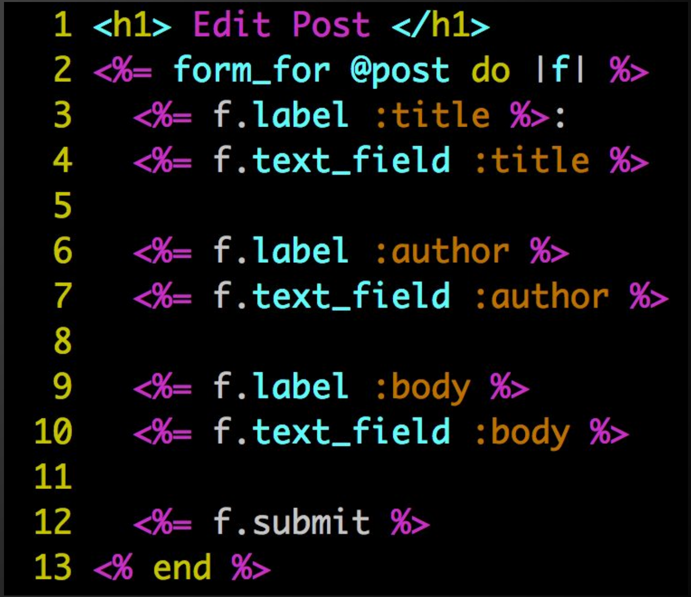
This will set @post in the view to the post we are intending to edit

In app/views/posts/index.html.erb

```erb
<%= link_to 'Edit post', edit_post_path(post) %>
```

Should be added directly under show post

app/views/posts/edit.html.erb

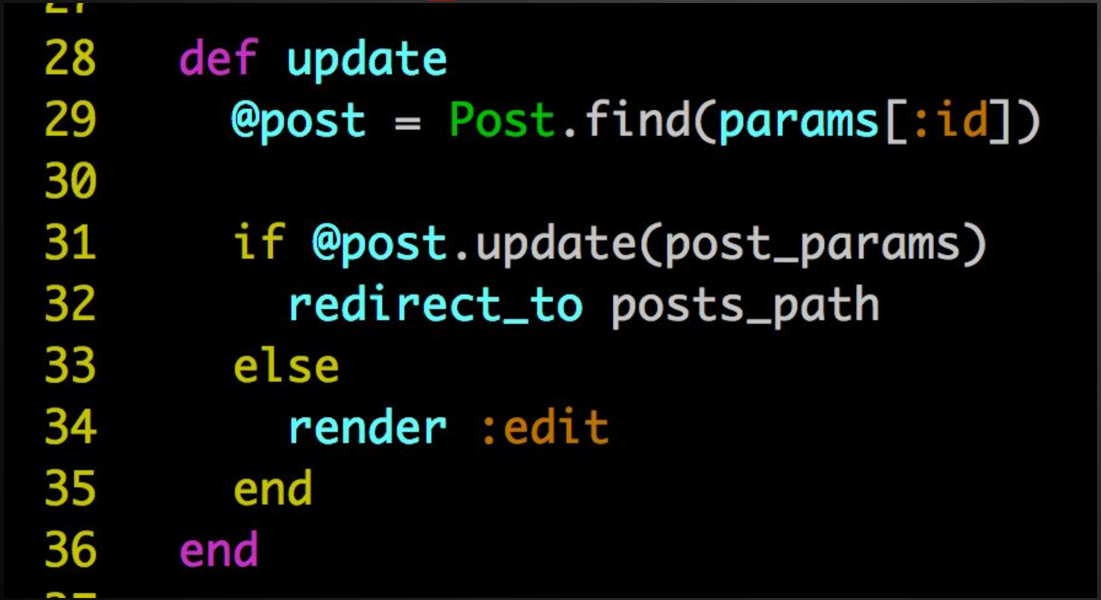


```erb
<h1> Edit Post </h1>
<%= form_for @post do |f| %>
  <%= f.label :title %>:
  <%= f.text_field :title %>

  <%= f.label :author %>
  <%= f.text_field :author %>

  <%= f.label :body %>
  <%= f.text_field :body %>

  <%= f.submit %>
<% end %>
```

Editing a post (cont)

Finally update the controller to handle the edit form submit

app/controller/posts_controller.rb



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

    if @post.update(post_params)
      redirect_to posts_path
    else
      render :edit
    end
  end
```

Test your code &

Delete

We almost have a full CRUD app we just need to be able to delete.

First update the post index view

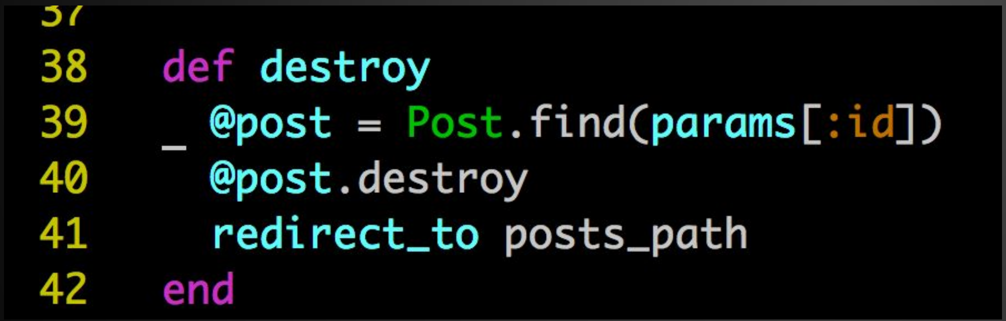
app/views/posts/index.html.erb

```
<h1> Posts </h1>
<p><%= link_to 'New Post', new_post_path %></p>
<% @posts.each do |post| %>
  <h2><%= post.title %></h2>
  <h3><%= post.author %></h3>
  <p><%= post.body %></p>
  <%= link_to 'Show post', post_path(post) %>
  <%= link_to 'Edit post', edit_post_path(post) %>
  <%= link_to 'Destroy post', post, method: :delete %>
<% end %>
```

Delete (cont)

Now create a destroy method in the controller

app/controllers/posts_controller.rb



```ruby
  def destroy
    @post = Post.find(params[:id])
    @post.destroy
    redirect_to posts_path
  end
```

.destroy removes the record from the database

Every method either needs to return HTML, a JSON object, or redirect

We destroyed the object and there is no HTML so in this case just redirect the user back to the posts/index route

Congrats you just built Facebook

The only thing left to do is….