Yalies on Rails - Week 2

Part 2: Basics of Rails

In these notes, we'll cover the basics of Rails apps, and convert our `lolcat.rb` program to a simple Rails app.

Version Check

First make sure you have Ruby installed, by running the following commands.

Remember: on Windows, you'll need to make sure you start your command prompt with Ruby and Rails loaded, so from the start menu, search 'rails' and choose **Command Prompt with Ruby and Rails**.

In a terminal, you should see something like the following:

```
$ ruby -v
ruby 1.9.2p320
```

Also check your Rails version:

```
$ rails -v
Rails 3.2.8
```

For Ruby, you need at least 1.8.7, for Rails, at least 3.2. If you don't have those, please refer to Week 1's notes for instructions to upgrade.

If you have trouble, come to office hours or post a question on Piazza, and I'd be happy to help you out.

Step One: Create a new app

Switch to your code folder

Make sure you're in the folder where you want to keep your code. If not, use the **cd** command to change directories.

```
$ cd path/to/your/code
```

Remember, don't include the '$' when typing commands, it's just there to indicate a terminal prompt. And of course, the path to your code folder will be different.

Create the app

We'll tell Rails to create a new blank Rails app for us, called **lolcat**:

```
$ rails new lolcat
```

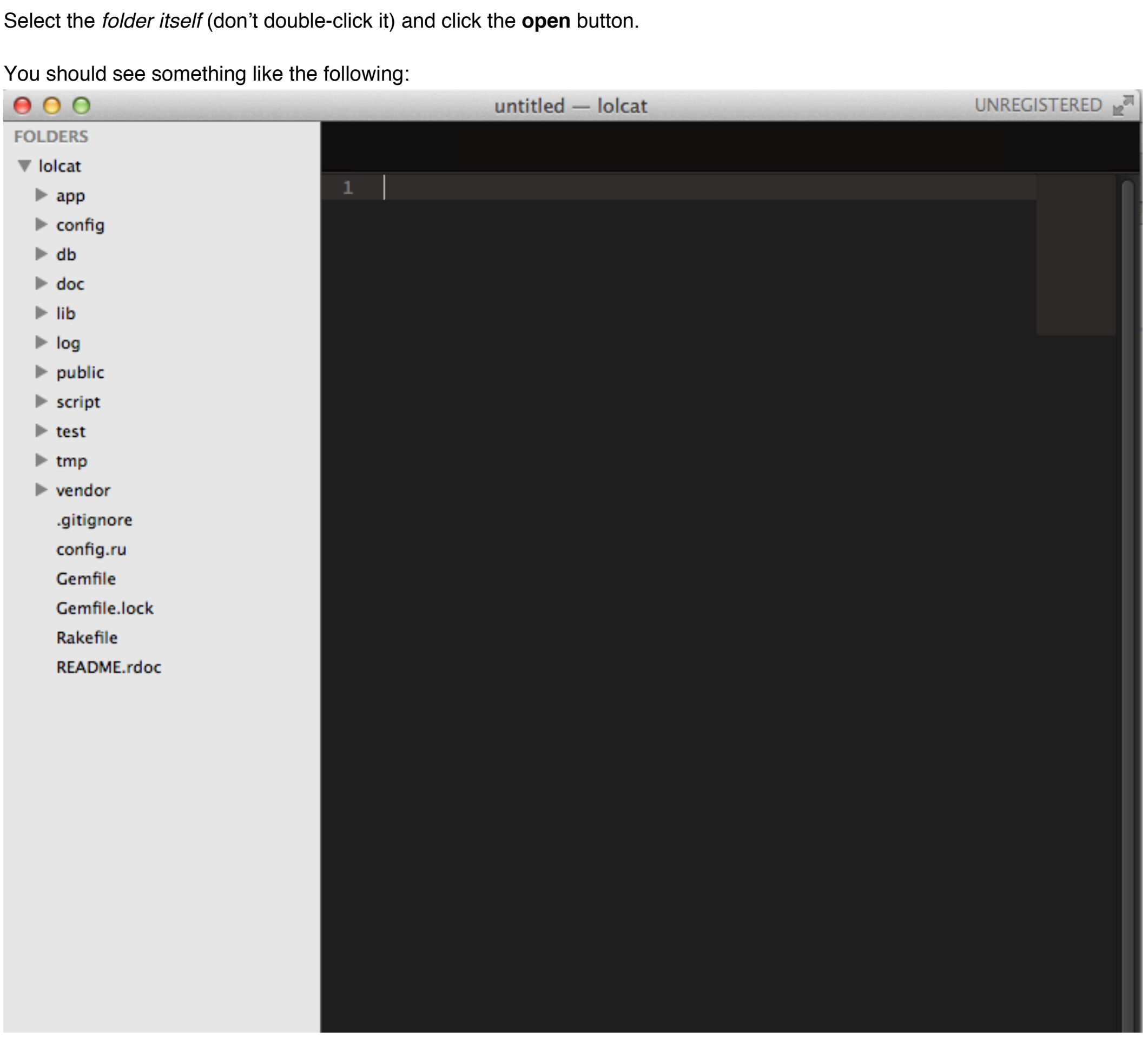
You'll see a bunch of text fly by, as Rails tells us all the folders and files it creates. When it's all done, you should see a new folder called **lolcat**, cd into it:

```
cd lolcat
```

Open in Sublime Text

Go ahead and open the app in Sublime Text. To do so, in Sublime Text choose File…Open. In the open dialog box, naviage to the lolcat folder. Select the *folder itself* (don't double-click it) and click the **open** button.

You should see something like the following:



A Detour: The Structure of Rails Apps

Before we continue, it's important to know a bit about how a rails app is laid out.

In the root folder of your app (in this case, the *lolcat* folder), you'll see a number of files and folders. Here's a quick rundown of what some of them do:

- **Gemfile** - If our app uses other gems (plugins), we list them in here. We won't use this yet though.
- **app** - Stores almost all of the code for our app that we write.
- **config** - Stores configurations files for our app.
- **db** - Most apps use a database. We store our database-related files here.

If you look inside the **app** folder, you'll see a few more folders. The three most important are the folders called **models**, **views**, and **controllers**.

Models, Views, and Controllers

Definition

Rails uses a *design pattern* called MVC (Model/View/Controller) for most of it's work. MVC informs how we write and organize our code, and how it all fits together.

Before we explain how they work together, here's a summary of what each part does:

- **model** - represents the information (data) of the application and the rules to manipulate that data.
- **views** - represent the user interface of your application. HTML templates with embedded Ruby (erb)
- **controller** - provide the "glue" between models and views. Processes incoming requests from the web browser, works with the models to get that data, and passes the data on to the views for presentation.

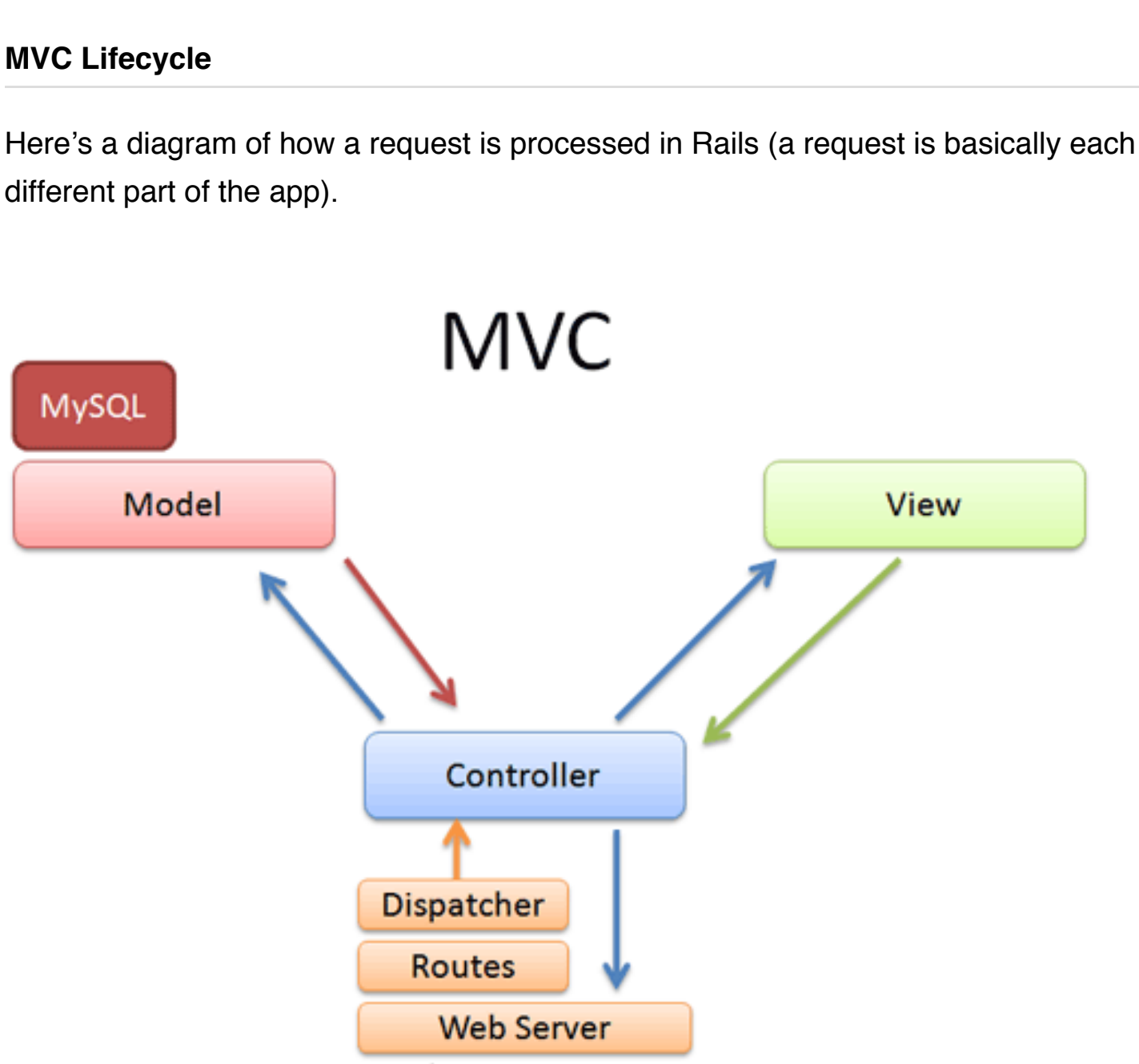
Rails also has a part called the **router**, not technically part of MVC, but very important:

- **router** - takes incoming requests and sends them to the correct controller for processing.

It's important to note that in a normal app, we'll have many different models, views, and controllers. In fact, we tend to create one controller and model for each *noun* in our app, and many views for each as well. We'll look at an example in a bit.

MVC Lifecycle

Here's a diagram of how a request is processed in Rails (a request is basically each time we visit our app in a browser, or click a link leading to a different part of the app).



A diagram of the Rails request lifecycle

A blog example

To explain this, let's consider the example of a simple blog. It has **Authors**, **Posts**, and **Comments**. In MVC, we would have one controller for each of those (author_controller.rb, post_controller.rb, and comment_controller.rb) and one model for each (authors.rb, posts.rb, comments.rb).

We'd also have a set of views for these as well. At the very least, we'd need:

- one for the main page of the blog (which is really just a set of posts)
- one to view a *specific* post (along with its comments)
- and one to manage authors

Let's say the user (an author for the blog) needs to edit a post. They clicks a link, `www.blogs4geese.com/posts/edit/going-south`. The way our blog works, this should take the user to a page where they can edit an existing post titled 'Going South'.

First, the request gets to our **router**. It looks at the URL we went to `www.blogs4geese.com/posts/edit/going-south` and breaks it down. It ignores the hostname, *www.blogs4geese.com*, and looks only at the path.

We've set up our router so that it maps the parts of the path in order to a controller, action, and a title. So `/posts/edit/going-south` starts up the **posts** controller, runs the **edit** action inside that controller, and passes along the title 'going-south'.

Inside our **edit** action, the controller knows it needs to talk to the `post.rb` model. It asks the model to find the post with the title 'going-south'. The model returns the relevant info, such as the author name, timestamp, and body of the post.

Finally, the controller gets the relevant view, in this case the view for editing a blog post, and substitues all the relevant info it got from the blog post model (title, content, timestamps, etc). It then returns this page back to the user's web browser for display.

And that's it! Time to take that and apply it to our lolcat example.

Step Two: Configure the Routes

Before we write our controller and views, we need to set up the router so it knows how to process our requests. For now, we just need to add one route.

Open up the `routes.rb` file (located in the config folder.) Edit it so that it looks like this:

```
Lolcat::Application.routes.draw do
    root :to => 'lolcat#index'
end
```

The first and last lines just tell Rails that this is our routes file. In between those two lines is where we put the important parts.

The line in the middle (beginning with `root`) configures our **root** route. This is the route that is matched when the path is *empty*. For example, if our app is hosted at *www.myapp.com*, this route will be matched when someone visits *www.myapp.com/* (with or without the final `/`).

In this case, we have set it to route to the **lolcat** controller, and the **index** method.

Step Three: Create the lolcat Controller

For our simple lolcat app, we'll only need a controller and a view. There's no data to store yet, so a model isn't necessary.

Create the Controller

Our controllers go in the `app/controllers/` folder. The file name must match the name of the controller. So, for our lolcat controller, the file would be named `lolcat_controller.rb`.

But instead of creating it ourselves, let's have Rails help us out. In a terminal (again make sure you `cd` to your app's root folder first), enter the following command:

```
$ rails generate controller lolcat
```

Rails will tell you what files it creates. In addition to the controller file, it will create a folder for our views, `app/views/lolcat/`, and some files related to testing our application and javascript (which we'll ignore for now).

Add the index Action

Every controller needs at least one **action**, which corresponds to a Ruby method. When we defined our routes, we specfied that the root route should go to the lolcat controller, and it's **index** action.

Let's go ahead and create our lolcat controller with the index action:

```
class LolcatController < ApplicationController

    def index
        @response = "oh hai!"
    end

end
```

All we're doing is creating a variable, called `@response`, and storing a string inside of it. As we'll see shortly, the `@response` variable is a special type of variable (technically it's an *instance* variable), since it starts with an `@`. In Rails, instance variables in the controller are accessible in our views.

Step Four: Create the lolcat index View

About Views

Views in Rails are templates. They define the layout and content of our application's interface, with fill-in-the-blank areas for us to customize. Thinking back to the blog example, we may define a template for what a blog post looks like, but we still need to fill in the blanks with the title, body, and author of a specific blog post when a user requests it.

In Rails, we build our views using HTML for the layout and structure, and embedded Ruby (erb) for the fill-in-the-blanks areas.

If you don't know HTML well, I suggest you take some time now to read a tutorial on HTML. I suggest the w3schools.com tutorial.

For erb, the main thing to know is that anything in between the special erb tags gets evaluated as Ruby. The erb tags are: `<% %>` and `<%= %>`. The only difference is that in the second version, the result of the Ruby code inside the tag is embedded into the HTML.

Create the index View

Let's go ahead and create our first view. It should be called `index.html.erb`, and you should save it in `app/views/lolcat/`.

It should look like this:

```
<h1>Lolcat Simulator 2001 ME</h1>
<h2>Lolcat saiz:</h2>
    <p>
        <%= @response %>
    </p>
```

Note the appearance of the `@response` variable, inside of the erb tags. We're telling Rails to embed the value of the `@response` variable into the resulting view sent back to the user.

Remove Rails' default view

While we're at it, let's remove an unnecessary view that rails created. In a terminal, enter the following command:

```
$ rm public/index.html
```

With that done, it's time to start our app for the first time!

Start the App!

Now that we've got the bare minimum created, it's time to start up our application. In your terminal, enter the following:

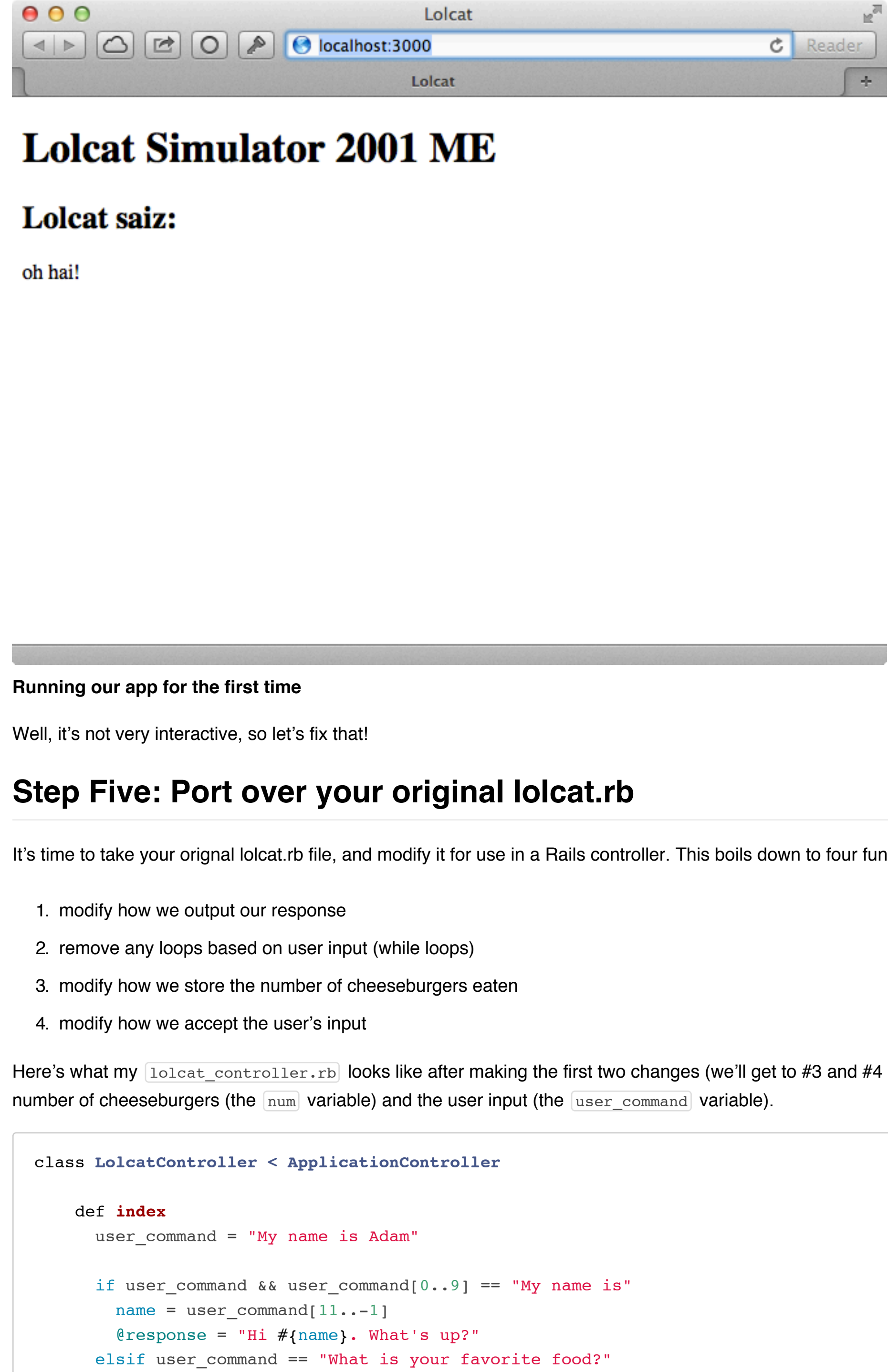
```
$ rails server
```

You'll see something like this:

```
=> Booting WEBrick
=> Rails 3.2.8 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-10-14 23:59:30] INFO  WEBrick 1.3.1
[2012-10-14 23:59:30] INFO  ruby 1.9.2 (2012-04-20) [x86_64-darwin11.4.0]
[2012-10-14 23:59:30] INFO  WEBrick::HTTPServer#start: pid=43376 port=3000
```

Give it a moment, and then switch to your web browser, and enter the following URL: `localhost:3000/`. This tells your browser to make a request to the Rails server running on your computer.

If you've done everything correctly, you should see something like this:



Lolcat Simulator 2001 ME

Lolcat saiz:

oh hai!

Running our app for the first time

Well, it's not very interactive, so let's fix that!

Step Five: Port over your original lolcat.rb

It's time to take your orignal lolcat.rb file, and modify it for use in a Rails controller. This boils down to four fundamental changes:

1. modify how we output our response
2. remove any loops based on user input (while loops)
3. modify how we store the number of cheeseburgers eaten
4. modify how we accept the user's input

Here's what my `lolcat_controller.rb` looks like after making the first two changes (we'll get to #3 and #4 later). For now, I've hard coded in the number of cheeseburgers (the `num` variable) and the user input (the `user_command` variable).

```
class LolcatController < ApplicationController

    def index
      user_command = "My name is Adam"

      if user_command && user_command[0..9] == "My name is"
        name = user_command[11..-1]
        @response = "Hi #{name}. What's up?"
      elsif user_command == "What is your favorite food?"
        @response = "i luv cheezburger"
      elsif user_command ==  "Here, have a cheeseburger."
        num = 10
        @response = "omnomnom...i've eeetin #{num} cheezburgers today"
      elsif user_command == "How many cheeseburgers have you eaten today?"
        num = 10
        @response = "i iz so full! like #{num} cheezburgers!"
      elsif user_command == "Bye"
        @response = "kthnx bai!"
      else
        @response = "wat did u sai?"
      end

    end

end
```

For #1, note that I just store what I previous outputted using `puts` into the @response variable, since that's what gets rendered in the view. For #2, I have removed the loop entirely. Since web apps are request-driven, our application will run this index method every time the user submits a command or refreshes the page. Thus, no loop is needed.

We can now go back to our web browser, and verify that we see the expected responses as we change the value of the `user_command` variable.

Go ahead, try it!

Step Six: Introducing the Session

Before we proceed, make sure you know what a Ruby hash is. If you don't, read the Week 2 - Part 1 lecture notes.

You'll notice in the previous version of lolcat that we had no way of storing information between requests. Every time we refresh the browser, we run a totally new copy of the index action. If, for example, we tried to update the `num` variable by increasing it, that change would be saved. When we refresh, the `num` variable would be reset to its original value.

If we want to store information, we'll need to use the Rails `session` feature. Sessions are a way to store small bits of data specific to each user interacting with your application. This is perfect, since we want each user to interact with their own lolcat. If someone else tells their lolcat to eat a cheeseburger, we only expect the number of cheeseburgers to increase for them. Otherwise it might get confusing for our users.

Along the same lines, think of a catalog. We want to keep track of our users' shopping carts independently. When someone adds something to their cart, it shouldn't be added to anyone else's cart!

The `session` is just a Ruby hash, so to stuff in the session, just add a key/value pair and assign a value to it:

```
session[:num] = 0
```

Rails takes care of matching the appropriate session values for each user. So you don't have to keep track of users, just store and retrieve items in your session hash, and it works like magic. (If you're curious, this is done using web browser cookies.)

But there's one problem, if we've already stored something in our user's session, we don't want to over-write it on each new request. To get around that, we use the following idiom:

```
session[:num] ||= 0
```

This essentially means, if `session[:num_cheeseburgers]` doesn't exist, create it and set it to `0`; if it does exist, keep its current value.

So, try modifying lolcat_controller.rb before looking at the code below. When you're done, compare it to the following:

```
class LolcatController < ApplicationController

    def index
      user_command = "My name is Adam"
      session[:num] ||= 0

      if user_command && user_command[0..9] == "My name is"
        name = user_command[11..-1]
        @response = "Hi #{name}. What's up?"
      elsif user_command == "What is your favorite food?"
        @response = "i luv cheezburger"
      elsif user_command ==  "Here, have a cheeseburger."
        session[:num] += 1
        @response = "omnomnom...i've eeetin #{session[:num]} cheezburgers today"
      elsif user_command == "How many cheeseburgers have you eaten today?"
        @response = "i iz so full! like #{session[:num]} cheezburgers!"
      elsif user_command == "Bye"
        @response = "kthnx bai!"
      else
        @response = "wat did u sai?"
      end

    end

end
```

Test everything, and make sure your lolcat knows how to keep track of cheeseburgers appropriately. If you want to see how the sessions support multiple users of your app, try opening the app in a separate browser. This creates a separate session, so you should be able to see how the cheeseburgers are independent for each browser.

Step Seven: Make It Interactive

Our last step is to allow our users to input commands. Admittedly, we've saved the most important for last!

Adding an HTML Form

To accept user input, we'll need to add an HTML form to our view. HTML forms consist of input elements (think text boxes, check boxes, submit buttons) and a location (URL) to submit that data to.

In our simple case, we'll have a form with one text box (for the user_command), and a submit button.

We could hand code these forms, but Rails gives us some helper commands to make it a bit easer. Add the following block of code to the end of `app/views/lolcat/index.html.erb`:

```
<h2>Sai sumthing elz:</h2>
    <%= form_tag do %>
        <%= text_field_tag 'user_command', 'enter command here...' %>
        <%= submit_tag 'commit' %>
    <% end %>
```

Let's look at this line by line (skipping the simple `<h2>` heading.)

The first line declares that we're creating a form. Since we don't specify a location to submit the form, it will default to submitting back to the current page (in this case, that corresponds to the lolcat controller's index action.)

The second line adds a text field (where the user will enter the command.) The `'user_command'` tells rails what to call the value of this text field so we can refer to it's contents back in our controller. The last part gives a default value.

The third line creates a submit button labeled 'submit'. When the user clicks this button, the form will be submitted to the designated location (again, in this case, it defaults to the current page.)

Finally, we declare that we've reached the end of the form.

Using Form Data in Our Controller

Our last change is to take the results of the form submission, and use it in our controller for our `user_command`.

When a user submits a form, Rails takes the data, and places it in a special hash, called `params`. Each element (text box, checkbox, etc.) in the form creates a key/value pair in the `params` hash, where the key is the name of the form element.

Since we named our text field 'user_command' (in the previous form), we can access the data for that text field by using `params[:user_command]`.

So, the final form of our controller should look like this:

```
class LolcatController < ApplicationController

  def index
    user_command = params[:user_command]
    session[:num] ||= 0

    if user_command && user_command[0..9] == "My name is"
      name = user_command[11..-1]
      @response = "Hi #{name}. What's up?"
    elsif user_command == "What is your favorite food?"
      @response = "i luv cheezburger"
    elsif user_command ==  "Here, have a cheeseburger."
      session[:num] += 1
      @response = "omnomnom...i've eeetin #{session[:num]} cheezburgers today"
    elsif user_command == "How many cheeseburgers have you eaten today?"
      @response = "i iz so full! like #{session[:num]} cheezburgers!"
    elsif user_command == "Bye"
      @response = "kthnx bai!"
    else
      @response = "wat did u sai?"
    end

  end

end
```

And there you have it! In just a few simple steps, we were able to port our lolcat.rb program to a fully-functioning Rails app.