INFO 2310

Topics in Web Programming
Ruby on Rails

Last Week on INFO 2310

Discussed some of the philosophies behind Rails:

* Agile
* Convention over Configuration
* MVC
* DRY

Set up our development environment

* An EC2 virtual machine
    * Ruby, Rails, Git, Heroku
* WinSCP and Putty to connect to it

Made the initial commit to our version control system on Github

Deployed our first Rails app to Heroku

? From last week

- Pass/Fail only (no grade option)
- Free instance hours
  - plus your AWS Credit
- Why do we need Heroku if we have EC2?

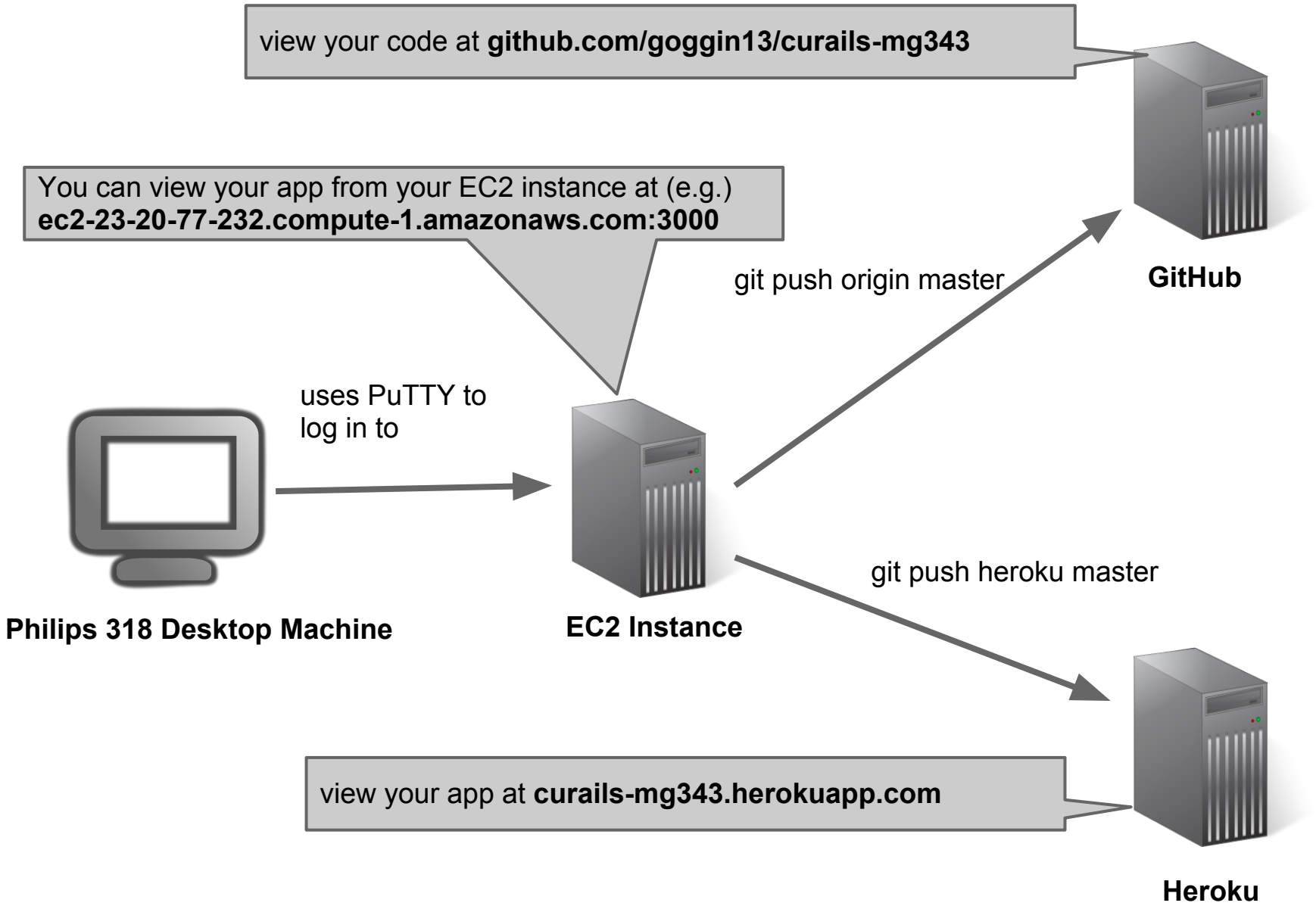
While we're chatting about these, go ahead and log into Amazon and start up your instance

Heroku vs. EC2

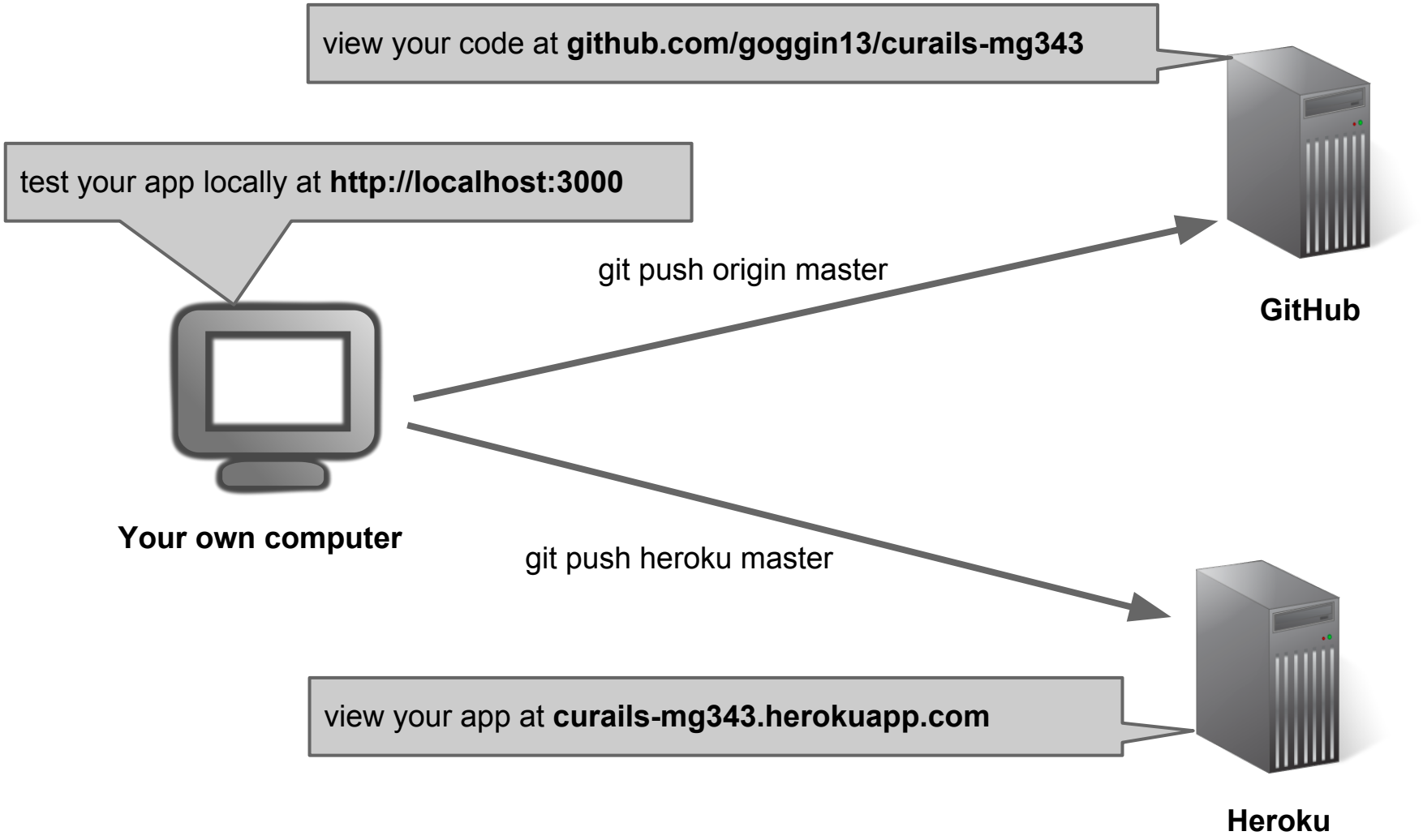
Heroku is

- easier to use
- free at very slow scale
- more expensive at larger scale
- less configurable

Our Dev environment



In more "normal" circumstances, the environments may look more like...



Git

First, we created an empty repository in the root of our Rails app.
git init

Then we added all of the files to the repository
git add -A

At this point, the files are not committed; they are "staged" for a commit
To commit them;
git commit -m "a really informative commit message"

Why are those two steps?

Git...

At this point, nothing is backed up on a different server. If your EC2 instance were to crash and burn, you would lose your code and be sad. Additionally, while you have committed code to your *local* git repository, if you were working with a team, none of them could see your changes yet.

git push origin master

git push <which_remote_server_to_push_to> <branch_name>

Now, we have pushed the "master" branch to the "origin" server, which we defined to be our GitHub repository.

(remember **git remote add origin <your_github_url>**?)

Now our code is backed up, and if we were working with other developers, they would be able to pull down our updated code.

Week 2

Static Pages &&
Rendering &&
Ruby

Less talking, more coding this time

Login to PuTTY

- open PuTTY
- On the left panel, navigate to Connection->SSH->Auth
  - Next to "Private key file for authentication:", click "Browse", and select the *.ppk file you created on the previous step.
- Then, navigate to Connection->Data
  - For "Auto-login username", type "ec2-user"
- Navigate to "Session" (the very top)
  - Copy the "Public Domain Name" of your EC2 instance; you can see this on the instances page of the Amazon console, when an instance is selected
  - Paste it into the "Host Name (or IP address)" field
- Click "Open"

Login to WinSCP

- Open WinSCP
  - Paste in your domain to "Host name", as you did in PuTTY
  - Type "ec2-user" for the "User name"
  - Click "..." to select your private key file
  - Click "Login"
- Set NotePad++ as the default editor.
  - Click Options->Preferences
  - Select "Editors" from the left tab
  - Click Add
  - Select "External Editor"
  - Find NotePad++ (C:\Program Files (x86)\Notepad++)
  - Click "Open", then "Okay"
  - Drag it to the top of the editor list

Branching

Before we make any code changes, its good practice to always do your work on a separate git "branch".

Once you've logged into Putty, you can accomplish this with these commands:

```
cd ~/apps/info2310                # change directories into your application
git checkout -b static_pages      # checkout a new branch named static_pages
git branch                        # view the current branches
```

At the end of the class, if all goes well, we will "merge" our static_pages branch with the master branch, and then deploy.

Routing in Rails

First, start your Rails server by typing

rails s

Last time we modified public/index.html. Recall you can see this file in the browser at

your_ec2_domain_name.com:3000/index.html

Anything in the public directory is served directly from there, without any interaction from rails.

Try visiting

your_ec2_domain_name.com:3000/robots.txt

This file is also served directly from the public/

Making our own static file

Using WinSCP, add a new file to **public/** named **hello.html**, with the following content

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

Then visit your_domain:3000/hello.html

You can then remove the file, since it doesn't really seem that useful

Slightly Dynamic Pages

Now let's start to actually use Rails; we'll create 3 static pages, Home, Help, and About.

We'll let Rails generate the first 2 for us, and we'll do the 3rd by hand.

rails generate controller StaticPages home help --no-test-framework

--no-test-framework tells rails to not generate the test files for these pages (don't worry, we'll get there)

Output (woah)

```
create  app/controllers/static_pages_controller.rb
 route  get "static_pages/help"
 route  get "static_pages/home"
invoke  erb
create    app/views/static_pages
create    app/views/static_pages/home.html.erb
create    app/views/static_pages/help.html.erb
invoke  helper
create    app/helpers/static_pages_helper.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/static_pages.js.coffee
invoke    scss
create      app/assets/stylesheets/static_pages.css.scss
```

What did we make?

You can now visit the following pages in your browser

your_domain.com:3000/static_pages/help

your_domain.com:3000/static_pages/home

Understanding the Magic

in the file config/routes.rb:

these lines...
get "static_pages/home"
get "static_pages/help"

Tell Rails that we want to route incoming HTTP GET requests for the path "static_pages/home" to the "home" function on the "StaticPagesController" class.

This is another example of convention over configuration; rails assumes the url is of the form "controller/action". If we did not want to follow this, we could be more verbose:

get "static_pages/home" => "foo#bar"

If you open up the file at

app/controllers/static_pages_controller.rb, you can see the two functions we just created

```
def home
end

def help
end
```

Why isn't there anything here...? Convention over configuration!

Rails assumes the function at "StaticPagesController#home" wants to render the view at "app/views/static_pages/home.html.erb". Should we want a different one we just override it...

```
def home
  render 'foo/bar.html.erb'   # render the template at app/views/foo/bar.html.erb
end
```

And the views

If we open up the files at
app/views/static_pages/home.html.erb
app/views/static_pages/help.html.erb

We can see the content that is being displayed in the browser

MVC

... or just VC, in this case

It may help you in understanding where these files live, and how they relate, to frame them in terms of the Model-View-Controller framework.

config/routes.rb

The routes we defined in config/routes.rb are for mapping a request to a specific controller action.

app/controllers/static_pages_controller.rb

The logic of the controller functions.

In our case, these are just blank function definitions. Later of course they will become more complicated, doing things such as authentication, and fetching models to return to the view.

app/views/static_pages/home.html.erb
app/views/static_pages/help.html.erb

The HTML served up to the user as the presentation layer

Another Philosophy

Test-Driven-Development

You should write tests *before* you write the code. Your development cycle should always* look like this:

1) Write a failing test

2) Write just enough code to make the test pass

3) Refactor to keep the code clean (while keeping the test passing)

and Repeat

This cycle is commonly referred to as "Red, Green, Refactor".

Our first tests

We'll use RSpec and Rails generators to create tests for our existing pages

We'll start with something called an integration test (as opposed to a unit test). An integration test simulates the actions of a user interacting with our application using a web browser.

```
# first, tell Rails to use the RSpec gem for testing
rails generate rspec:install

# then generate the tests for our controller
rails generate integration_test static_pages
```

Open up the file at spec/requests/static_pages_spec.rb
(requests is what RSpec calls integration tests)

Don't worry if the generated content looks meaningless to you!
Replace the contents of the file with this

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'INFO 2310 MicroPoster'" do
      visit '/static_pages/home'
      page.should have_content('INFO 2310 MicroPoster')
    end
  end
end
```

RSpec is a Domain Specific Language, intended to make writing tests look a little more like English. Reading through the code, it should be fairly clear what we are testing.

```
# We want to visit the home page
visit '/static_pages/home'

# And make sure the content we expect is there
page.should have_content('INFO 2310 MicroPoster')
```

Running the test (Red)

Type the following in the terminal to run the tests:

bundle exec rspec spec/requests/static_pages_spec.rb

After a brief pause, RSpec will report a failure. Which is just we want to start our "Red, Green, Refactor" cycle.

Green

To get the test to pass, let's edit
app/views/static_pages/home.html.erb
to the following:

```
<h1>INFO 2310 MicroPoster</h1>
<p>
  Welcome to the INFO 2310 MicroPoster App.
</p>
```

Run
bundle exec rspec spec/requests/static_pages_spec.rb
again to see the tests pass.

Red

Now let's add the analogous test for the Help page to spec/requests/static_pages_spec.rb

```
describe "Static pages" do
  ....
  describe "Help page" do

    it "should have the content 'Help'" do
      visit '/static_pages/help'
      page.should have_content('Help')
    end
  end
end
```

And check that the tests fail:
bundle exec rspec spec/requests/static_pages_spec.rb

Green

To get the test to pass, let's edit
app/views/static_pages/help.html.erb
to the following:

```
<h1>Help</h1>
<p>
  Get help on the INFO2310 MicroPoster at
  <a href="http://curails.herokuapp.com">the course website</a>
</p>
```

Run
bundle exec rspec spec/requests/static_pages_spec.rb
again to see the tests pass.

Red

Now a test for the (currently non-existent) About page
spec/requests/static_pages_spec.rb

```
describe "Static pages" do
  ....
  describe "About page" do

    it "should have the content 'About'" do
      visit '/static_pages/about'
      page.should have_content("About")
    end
  end
end
```

And check that the tests fail:
bundle exec rspec spec/requests/static_pages_spec.rb

Red (again)

The first error we get is
No route matches [GET] "/static_pages/about"

To fix this, we need to edit the file at
config/routes.rb
to include a new route for our page

```
Info2310::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  get "static_pages/about"

  ...
end
```

Red (and again)

Running the tests again shows us
The action 'about' could not be found for StaticPagesController

To fix this, we add an "about" method to our controller at
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

Red (still)

Running the tests again shows us
ActionView::MissingTemplate:
  Missing template static_pages/about

To fix this, we create a new page at
app/views/static_pages/about.html.erb

```
<h1>About</h1>
<p>
  Like Twitter... but better
</p>
```

And now... we should be back to green

Refactor?

Our application so far is too small to develop any real code smells we should be combating.

But don't worry

It will

Some small test improvements

You've mastered the "have_content" method. But it's quite unspecific. Let's be a little more precise.

Let's update each of our tests that look like:

```
it "should have the content 'Help'" do
  visit '/static_pages/help'
  page.should have_content('Help')
end
```

to

```
it "should have the h1 'Help'" do
  visit '/static_pages/help'
  page.should have_selector('h1', text: 'Help')
end
```

Still green?

Run the tests again to be sure they still pass

Back to Red

Our next task is to set the title element of each page to be something like

"INFO2310 Microposter | Home" and
"INFO2310 Microposter | About" etc.

Let's add a test like this to each of our 3 page test blocks

```
it "should have the title 'Home'" do
  visit '/static_pages/home'
  page.should have_selector('title',
                            :text => "INFO2310 Microposter | Home")
end
```

The complete code for the static_pages_spec file is listed in the lecture_2.txt file with all of the other code snippets.

Go ahead and run the tests to be sure they are failing

Layouts

Take a look at
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Info2310</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

What is it?

By default, Rails uses this file to render the layout for all the other views we render.

You may create other layouts and tell Rails to use them instead; but often a single layout for your site will do just fine.

The contents of individual pages are placed into this line

<%= yield %>

You need not concern yourself with the sorcery behind this quite yet.

A dynamic title

Let's edit the title to do what we need:

<title>Info2310</title>

becomes

<title>INFO2310 Microposter | <%= yield(:title) %></title>

What is this?

This means that every view which uses this layout is expected to provide a value for 'title' in the following fashion:

<% provide(:title, 'Home') %>

Again; don't be too concerned with the precise machinery of this yet.

Update our views

To each of our views:

app/views/static_pages/home.html.erb

app/views/static_pages/help.html.erb

app/views/static_pages/about.html.erb

lets add the appropriate lines to the top of them

<% provide(:title, 'Home') %>

<% provide(:title, 'Help') %>

<% provide(:title, 'About') %>

You can see the full code for each of the view files in lecture_2.txt

Now our tests should pass

Sanity check

Tests are great, but makes some sense to validate this is showing up in the browser as well.

Lets run

rails s # this is an alias for 'rails server'

in our PuTTY terminal, and then visit

your_domain.com:3000/home

your_domain.com:3000/help

your_domain.com:3000/about

A title helper

app/helpers/application_helper.rb

```
module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title)
    base_title = "INFO2310 Microposter"
    if page_title.empty?
      base_title
    else
      "#{base_title} | #{page_title}"
    end
  end
end
```

And update application.html.erb

```
<!DOCTYPE html>
<html>
<head>
<title><%= full_title(yield(:title)) %></title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

Seems like a good place to commit

```
git status                          # see wassup
git add -A                          # add everything

# Commit
git commit -m "home, help, and about page"
```

Remember, in proper Git form, we were working on a feature branch; which we can now merge back into master

```
git checkout master          # checkout the master branch
git merge static_pages           # merge in changes from the feature branch
git branch -d static_pages   # delete the static pages branch
git push origin master       # push to github
git push heroku master       # deploy to heroku
```

Now... let's talk Ruby

In PuTTY, type
rails console

This opens up a prompt, with all of your application code loaded. It is an EXTREMELY useful tool for testing, debugging, and experimenting.

Hello World

puts "hello, world"

puts is a Ruby function that has the side effect of printing its argument to the screen.

Note that

- The function returns 'nil' which is Ruby for nothing (a la **null** in Java).
- When calling functions in Ruby, the parenthesis are optional
  - This is equivalent to **puts("hello, world")**

Strings

```
"hello"
"hello, " + "world"
first_name = "Matt"
"#{first_name} Goggin"
last_name = "Goggin"
first_name + " " + last_name
"#{first_name} #{last_name}"
```

Kinds of strings

Use single quoted strings when you want the string to contain exactly what you typed

```
x = "this string has an actual \n newline in it"
puts x
x = 'this string has an escaped \n newline in it'
puts x

my_name = "Matt"
puts "my name is #{my_name}"
puts 'my name is #{my_name}'
```

Objects

In Ruby, **everything** is an object. Objects respond to messages

```
# Strings are objects
"hello".length

# nil is an object
nil.nil?
nil.to_s

# integers are objects
1.even?
27.gcd(36)                    # seriously ruby? you rock
11.next
```

Branching

```
x = "hello"
if x.empty?
  puts "x is empty"
else
  puts "x is not empty"
end

unless x.empty?
  puts "x is not empty"
end

puts "x is empty" if x.empty?
puts "x has content" unless x.empty?
```

Truthyness

In Ruby, the only items that return false are the boolean value **false** and **nil**

```
puts "false is false" if false
puts "nil is false" if nil
puts "0 is not false" if 0
puts "the empty string is not false" if ""
```

method definitions

```
def string_message(string)
  if string.empty?
    "It's an empty string!"
  else
    "The string is nonempty."
  end
end
```

note no explicit return statements are required

Arrays

```
[1, 2, "hello", "world"]
"fooxbarxbaz".split("x")
a = [14, 31, 17, 45]
a << 42
a[0]
a[-1]
a[2..4]
a.first
a.second
a.last
a.length
a.sort
a.shuffle
a.reverse   # these don't modify a
a.reverse!  # these do
```

Hashes

```
user = {}
user["name"] = "goggin13"
user["name"]
user = {"first_name" => "matt", "last_name" => "goggin"}
```

it's not actually common to use strings as hash keys; symbols are preferred

```
user = {:first_name => "matt", :last_name => "goggin"}
user = {first_name: "matt", last_name: "goggin"}
```

A symbol is like a string; but it's immutable and has fewer methods (there's a bit more to it than that, but that will suffice for now)

You will see them used all over in Rails code, especially as hash keys

Blocks

Lots (and lots) of objects in Ruby respond to functions which take blocks as arguments.

For example

```
3.times { puts "hello, world" }

user = {:first_name => "matt", :last_name => "goggin"}

user.each do |key, value|
  puts "#{key} => #{value}"
end

user.each { |key, value| puts "#{key} => #{value}" }
```

The convention is to use curly braces "{}" for blocks that fit on a single line, and "do...end" for blocks that span multiple lines

More blocks

```
[1,2,3].map { |i| i * 2 }

sum = [1,2,3,4].inject(0) do |acc, i|
      acc + i
end

# in Rails
User.all.each do |user|
      puts "a user named #{user.name}"
     puts "*" * 80
end
```

Defining classes

```
class Word
  def palindrome?(string)
    string == string.reverse
  end
end

w = Word.new
w.palindrome?("foobar")
w.palindrome?("level")
```

Inheritance

```
class Word < String
  def palindrome?
    self == self.reverse
  end
end

w = Word.new("level")
w.palindrome?
```

Monkey Patching

```
class String
  def palindrome?
    self == self.reverse
  end
end

"level".palindrome?
```

make sure you have a good (really REALLY good) reason to do this

That's enough (too much) for today

Today we...

- Built our first static pages
- Made them slightly more dynamic, using test driven development
- Wrote our first helper
- Played with Ruby!

Check out the Hartl chapter on Ruby

http://ruby.railstutorial.org/chapters/rails-flavored-ruby

Stop your instances

Remember to stop (but NOT terminate) your EC2 instance by right clicking on it, and choosing "stop".

You will be able to reboot next time from where we left off, and you won't be charged while the instance is not running.

See you next week!