Informe de la práctica de Laboratorio Nº6: Repasando Ruby para Rails y aprendiendo Python para Django.

Autor: Adán Rafael López Lecuona.

Ruby para Rails

Introduciremos la generación dinámica de contenido HTML utilizando Ruby y aprenderemos a añadir algo de diseño utilizando CSS

Para empezar empezamos por una vista que sirve para eliminar las vistas duplicadas:

El archivo está dentro de /views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></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>
```

El título es tomado de cada vista como por ejemplo el home:

```
<% provide(:title, 'Home') %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Con la etiqueta primera del código anterior.

Creamos un full tittle para las páginas que no tienen título ponerle un título base.

Lo hacemos en helpers/application_helper.rb

```
module ApplicationHelper


  def full_title(page_title)
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
```

```
    else
      "#{base_title} | #{page_title}"
    end
  end
end
```

Ahora podemos cambiar los títulos de nuestras páginas.

```
<title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
```

Ya no hace falta tenerlo escrito ahí, lo sustituimos por esto:

```
<title><%= full_title(yield(:title)) %></title>
```

Tenemos que actualizar nuestros tests para la página de Inicio:

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      expect(page).to have_content('Sample App')
    end

    it "should have the base title" do
      visit '/static_pages/home'
      expect(page).to have_title("Ruby on Rails Tutorial Sample App")
    end

    it "should not have a custom page title" do
      visit '/static_pages/home'
      expect(page).not_to have_title('| Home')
    end
  end
  .
  .
  .
end
```

Para que los tests pasen correctamente tenemos que borrar la línea de provide del archivo home.html.erb dejándolo así:

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
```

```
    sample application.
  </p>
```

Ahora si ejecutamos nuestros tests :

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

Ahora veremos ejemplos con strings y métodos en Ruby. Para ello abrimos la consola de rails.

```
$ rails console
Loading development environment
>>
```

Los comentarios se escribe seguidos del símbolo #.

Las cadenas pueden estar entre comillas dobles("...") o comillas simples (’...’).
Ejemplo con cadenas:

```
$ rails c
>> ""
=> ""
>> "foo"
=> "foo"
```

Podemos concatenar las cadenas con el operador +.

```
>> "foo" + "bar"
=> "foobar"
```

Se imprime mediante puts:

```
>> puts "foo"
foo
=> nil
```

Veremos cómo tratar a los objetos en Ruby.

Un ejemplo de usar los objetos combinando operadores:

```
>> x = "foo"
=> "foo"
>> y = ""
=> ""
>> puts "Both strings are empty" if x.empty? && y.empty?
=> nil
>> puts "One of the strings is empty" if x.empty? || y.empty?
"One of the strings is empty"
=> nil
>> puts "x is not empty" if !x.empty?
"x is not empty"
=> nil
```

En Ruby, se llama a un método con la notación punto. El objeto con el que nos comunicamos se nombra a la izquierda del punto.

La definición de métodos en la consola también se puede realizar. Un ejemplo de esto lo definimos a continuación:

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

Se pueden crear un array listando elementos entre corchetes ([ ]) y separándolos por comas. Los arrays en Ruby pueden almacenar objetos de diferentes tipos.
Los arrays se pueden concatenar y repetir, igual que las cadenas.
Hicimos numerosos ejemplos con ellos:

```
>>  "foo bar     baz".split
=> ["foo", "bar", "baz"]
```

Otro ejemplo con arrays:

```
>> a = [42, 8, 17]
=> [42, 8, 17]
>> a[0]
=> 42
>> a[1]
=> 8
>> a[2]
=> 17
>> a[-1]
=> 17
>> a
=> [42, 8, 17]
>> a.first
=> 42
>> a.second
=> 8
>> a.last
=> 17
>> a.last == a[-1]
=> true
>> x = a.length
=> 3
>> x == 3
=> true
>> x == 1
```

```
=> false
>> x != 1
=> true
>> x >= 1
=> true
>> x < 1
=> false
>> a
=> [42, 8, 17]
>> a.sort
=> [8, 17, 42]
>> a.reverse
=> [17, 8, 42]
>> a.shuffle
=> [17, 42, 8]
>> a
=> [42, 8, 17]
>> a
=> [42, 8, 17]
>> a.sort!
=> [8, 17, 42]
>> a
=> [8, 17, 42]
>> a.push(6)
=> [42, 8, 17, 6]
>> a << 7
=> [42, 8, 17, 6, 7]
>> a << "foo" << "bar"
=> [42, 8, 17, 6, 7, "foo", "bar"]
>> a
=> [42, 8, 17, 7, "foo", "bar"]
>> a.join
=> "428177foobar"
>> a.join(', ')
=> "42, 8, 17, 7, foo, bar"
```

Un bloque es una porción de código encerrada entre paréntesis {} o entre do…end**.** Por lo tanto, un bloque es una forma de agrupar instrucciones, y solo puede aparecer después de usar un método: el bloque empieza en la misma línea que usa el método.

Un método puede usar el bloque mediante la palabra yield.

Veremos algún ejemplo de bloques de Ruby:

```
>> (1..5).each { |i| puts 2 * i }
2
4
6
8
10
=> 1..5
```

Los hashes de Ruby.Los hash se pueden crear mediante pares de elementos dentro de llaves({ }). Se usa la clave para encontrar algo en un hash de la misma forma que se utiliza el índice para encontrar algo en un array.

Un ejemplo de hashes:

```
>> user = {}
=> {}
>> user["first_name"] = "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"
=> "Hartl"
>> user["first_name"]
=> "Michael"
>> user
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

En Ruby, la definición de una clase es la región de código que se encuentra entre las palabras reservadas class y end.

Dentro de esta área, def inicia la definición de un método, que como se dijo en el capítulo anterior, corresponde con algún comportamiento específico de los objetos de esa clase.

Un ejemplo sencillo de constructor:

```
>> s = "foobar"
>> s.class
=> String
```

Con los arrays lo hacemos de la misma manera que con las cadenas:

```
>> a = Array.new([1, 3, 2])
=> [1, 3, 2]
```

Con los hashes se tienen que construir de diferente (0) manera que los anteriores:

```
>> h = Hash.new
=> {}
>> h[:foo]
=> nil
>> h = Hash.new(0)
=> {}
>> h[:foo]
=> 0
```

Después de hacer ejemplos mediante la consola.

Ahora incluimos hojas de estilo en cascada en nuestro diseño. Son similares las dos.

```
stylesheet_link_tag "application", { media: "all",
                                     "data-turbolinks-track" => true }
stylesheet_link_tag "application", media: "all",
                                   "data-turbolinks-track" => true
```

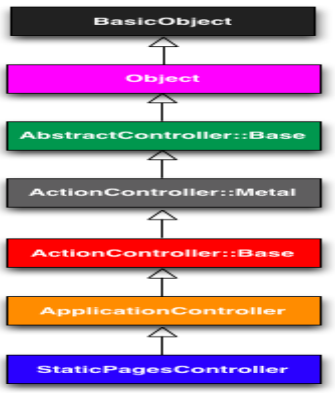
La fuente HTML producida por la CSS incluye:

```
<link data-turbolinks-track="true" href="/assets/application.css" media="all"
rel="stylesheet" />
```

Para ver el archivo CSS tenemos que ejecutar el servidor:

http://localhost:3000/assets/application.css

Jerarquía de el controlador StaticPages en Ruby:



Ahora veremos como cargar css en las plantillas de Django. Para introducir las páginas estáticas en Django debemos introducirlo en el archivo de configuración de la App:

```
STATIC_ROOT = ''
STATIC_URL = '/css/'
STATICFILES_DIRS = ("/css")
INSTALLED_APPS = (
    'django.contrib.staticfiles',
)
```

Modificar en urls.py:

```
urlpatterns = patterns('',
    (r'^$',home),
)
```

Modificar en views.py:

```
def homefun(request):
    return render_to_response('home.html')
```

Y la plantilla padre es base.html que carga el css, para eso introducimos en él lo siguiente:

```
<link rel="stylesheet" type="text/css" href="/css/style.css" />
```