**Sinatra Blog**

You've already made an object oriented terminal blog. Now it's time to bring (=llevar, traer) that technology to the Web. Our Sinatra blog **will still be object oriented** and it will follow the **model-view-controller** pattern. What's that you are saying? WordPress? Never heard (=oyó) of it!

**Model-view-controller**

Now that we are following the MVC pattern, the design of our application is going to change.

Even though (=aunque, a pesar de que) we are building this for the Web, we still want Post and Blog classes. Some of the functionality of those classes needs to change since displaying information to the user on the Web is different. In particular, any code that affects how the posts look for the user will be removed from the classes. How the information is shown to the users is the job of the **view**. Our classes are not our view. Sinatra has the views/ folder for our ERBs. Those will be our views.

Our Post and Blog classes represent the information that we are trying to manage, organize and display. In other words, our classes are acting as our **models**. Our classes and objects are kind of (=un poco, algo así, casi) like a digital representation (model) of that information.

**Writing tests for all the things**

Aside from making a Web version of this blog, the exercise will have you writing a lot of tests. Every feature that you add to the Post and Blog classes needs a test. If you add any additional classes, they need tests as well.

Remember to put your test suites (=juegos) in the spec/ folder and follow the something\_spec.rb naming convention. Each iteration will give you an indication of what you should be testing.

**Project structure**

You should be very organized for this project. Everything starts with a server.rb.

Create separate files for each of your classes in a lib/ folder. Pull (=sacar, extraer) those files into your server.rb with require\_relative.

All your test suites should go in the spec/ folder. Since we are starting off with (=empezando con) Post and Blog classes, at the very least you will need two files there.

Your ERB files should be in views/ so that they work with Sinatra. Don't forget your Gemfile!

sinatra\_blog/

├── Gemfile

├── lib/

│ ├── blog.rb

│ └── post.rb

├── server.rb

├── spec/

│ ├── blog\_spec.rb

│ └── post\_spec.rb

└── views/

└── ...

Anyway (=de todas formas, de todos modos), let's get to it.

**Iteration #1: List of posts**

The blog's homepage (=página de inicio) should be a list of posts in the blog. You will need your Blog class that allows you to access the list of posts. First create an instance of Blog and add posts to it. Perhaps with an *#add\_post* method on the Blog?

**# server.rb**

**# [...]**

blog = Blog.**new**

blog.add\_post Post.**new**(...)

blog.add\_post Post.**new**(...)

blog.add\_post Post.**new**(...)

Each instance of the Post class should store the post's **title**, **date** and **text**. The Post class will only be used to store and retrieve this information. That means you will need *#title*,*#date* and *#text* methods.

Your Blog class should have a *#posts* method that returns an array. That array should contain Post instances. With that array, you can loop over it in a view to show each post's title and date.

Things to test

1. That the Blog's *#posts* method returns an array of the posts added.

**Iteration #2: Showing latest posts first**

Next up, we want the posts to be listed in order of date, with the **latest posts being first**.

If we applied this change to the Blog *#posts* method, that would require us to change our tests.

Instead, let's **create a new method** on Blog for this new feature. Let's call it *#latest\_posts*. Now in your Sinatra route you will use *#latest\_posts* instead of *#posts* and everything should work perfectly. It still should return an array, after all!

Things to test

1. That the Blog's *#latest\_posts* method returns an array of the posts in the correct chronological order.

**Iteration #3: Post details page**

Now that our list of posts is in a good state, let's add the ability to actually (=de verdad, realmente) see the posts' text. Let's make it so that **clicking on the name of the post** will take the user to the **post details page**.

For that you need to add <a> tags to the post names that take the user to a new route.

<% @posts.each\_with\_index do |post, index| %>

<a href="????"><%= post.title %></a>

<% end %>

That new route should be a post details route. In that route you need to have **some sort of identifier in the URL** for the particular post you are referring to. How do you access an individual post from the array of posts? Through the **array index**! So the top post would be index 0, the second post would be index 1 and so on.

The URL to see the latest post could be something like /post\_details/0 and the URL to see the second post could be something like /post\_details/1. You need to make a route that can handle any number in the URL. You need to be able to see the details of any post in the blog.

On that post details route, you should show an ERB with the post's **title, date and its text**.

Things to test

**Nothing!** You should already have a test for the *#text* method, which is the only new thing we are using here from the Post class.

**Iteration #4: Creating posts**

Since this blog is Web-ready, we can make it even better than our terminal version by allowing users to **create new posts**.

Add a Create new post link to the home page that takes a user to a new route. That new route should show the user a form to create a new post on the blog. What would be a good URL for that route? We are showing a form to create a new post, so how about /new\_post?

To create an instance of Post, you need a title, a date and text. Do you need inputs for all three of those fields? Can you obtain one of those values without asking it from the user? Which input types should you use?

Remember there are **two routes** involved in every form transaction: **a GET and a POST**.

get "/new\_post" **do**

*# Show the form*

**end**

post "/????" **do**

*# Receive the form submission*

**end**

What would be a good URL for that POST route? Once you get your instance of the new blog post, what should you do with it?

Things to test

**Nothing!** All you need from your classes is to create a new Post and then add the newly created Post the the Blog. The Post class already let's you create a new one and the Blog class' *#add\_post* method will allow you to add it. We've already tested those.

**Iteration #5: Category and author**

Let's make our posts a bit more complex by adding a couple (=un par) more fields. Let's add post category and author. Both of these fields should be simple strings.

This change should only affect the Post class. The Post spec should be updated accordingly (=por lo tanto, por consiguiente).

Don't forget to **add inputs** for these two new fields to the **new post form** and the route that handles (=maneja) the form submission (=presentación, entrega, envío).

You'll also want to display the new fields on the post details page. Consider displaying one or both of them in the list of posts as well.

Things to test

1. The Post class' new *#category* method.
2. The Post class' new *#author* method.