Depot Application: Seller
CSC 340

The big-picture idea of the Depot Application is that it's a buying and selling website. You can imagine the buyer's view of the application, including a catalog page, a shopping cart page, a checkout page, and a receipt page. For the seller's view, we can consider a login page, a menu page, a page to create a new product to sell, a page to show that product, and a page to view pending orders.

In this worksheet, we will create the first part of our depot application – we will focus here on parts of the seller's view, and a listing of all available products.

1. Create a new Rails application called depot. Remember to CD to the depot directory before you continue to the next step.

2. Generate a scaffold for a Product resource. Product must have a title (of type string), a description (text), a price (decimal) and image_url (string). Image_url will give the directory of the image to be displayed in the catalog view.

3. Apply the migration and check your model - make sure everything is working properly.
   Now, let's add some validations to the model. Study the validations below:

   ```
   validates :price, :title, :description, :image_url,
             :presence => { :message => 'must be entered' }
   validates :price, :numericality => { :greater_than_or_equal_to => 0.01,
                                        :message => 'must be at least 0.01' }
   validates :image_url, :format => { :with => %r{\.(?:gif|jpg|png)\z},
                   :message => 'must be a URL for GIF, JPG, or PNG image' }
   ```

   Get comfortable with what they're doing – not that you've memorized the syntax, but that you can read it and see the purpose of it. Then add these validations to the model file.
   IMPORTANT NOTE: The code above should be all fine, but remember in general that when you copy code from a document (.docx, .pdf, etc.), you may need to double check that the quotes are simple quotes, not the "smart quotes" some document-creation software makes automatically. Smart quotes won't always be recognized appropriately in code files, and you'll get confusing errors.

4. Start the server and check your application. Note that the validation messages appear if you break any of the rules.

5. Now, it is time to add some test data to your application. We will create a migration that only adds test data.
   a. Create the migration using: rails generate migration add_test_data

   b. Copy the content of the add_test_data file from the I drive public folder for this worksheet. Add the content to the migration file you just created (overwriting the default contents of the file). Scan through the file to get an idea for how it works.
   c. Rake the migration
   d. Copy the necessary images from the public I: drive folder for this worksheet, and put them in your app/assets/images directory.
   e. Examine your application to make sure that everything is working properly. Note, however, that images won't show up on this page, and there are HTML tags in the descriptions.
      i. If you made a mistake and want to edit and rerun the migration, you can't just run it again: Rails will think it already did the migration and will ignore your request. So you'll have to enter `rake db:rollback` first, then `rake db:migrate` to run your newly edited migration.
   f. You'll note that the product listing doesn't execute the HTML tags in the descriptions. Add the `raw` method to the description field in index.html.erb to make the tags get executed:
      `<%= raw product.description %>`
      It's still all rather ugly, but `raw` helps a little bit.

6. Generate a controller called Store, with just one action called index.

7. In the index action of the store controller, write code to load all the Product objects into an array called @products.

8. Edit the view corresponding to the index action of the store controller so that the Products are displayed in a table.
   a. To start out, you'll need to make a table in HTML with an embedded Ruby loop through @products.
   b. Note that to see the index action's view, you'll need to visit http://.../store/index, not just http://.../store. This is because Store is not a resource built with scaffolding, so there's no route in routes.rb for the latter. Only the former is specified in routes.rb.
   c. Use the built-in Rails method number_to_currency to display the price using a currency format.
   d. Also use the image_tag method to display the images. The images will be found as long as they're in app/assets/images.

9. Let's make this look a little nicer.
   a. If you haven't already, remove the default text at the top of the store index view, and put a header that says "Available Products".
   b. Copy the following CSS into app/assets/stylesheets/store.css.scss:
```
table, td, th
{
        border:1px groove green;
}
th
{
        background-color:green;
        color:white;
}
```

We'll talk about CSS more soon, but you can probably tell that this will change all table, td, and th elements to have a border, and th elements will also have white text on a green background.

   c. Reload the store index view and observe the results.

10. Next, let's add a button that says "Add to cart". One of these buttons should be next to every Product object. When clicked, we want this button to add the object to a cart.
    a. You should use the button_to helper to do this since button_to generates a POST request and adding an object to the cart changes the state of the server. (GET requests are not supposed to try to change the state of the server.)
    b. So, in the index view of the store controller, add the line <%= button_to 'Add to Cart' %> after you display the cost of the Product object (in the same table cell). Of course, this button won't work yet – we'll get to that in a future worksheet.

11. In the Depot Application: Buyer worksheet, we will learn to use sessions to keep track of browser requests.