Depot Application: Buyer
CSC 340

In this worksheet, we will create the second part of our depot application, focusing here on the buyer's view.

1. In the previous worksheet, we named our application "depot". This will be the general name we continue to use for the application. But before you begin this worksheet, we should save what you did for the previous one, so you have a record of it. So, in the directory-tree area of Nitrous.IO, right-click on your depot directory and choose "Rename" to rename it depot1.

2. Next, go to the terminal (the bottom part of the window) in Nitrous.IO. Get into the directory that contains all of your projects. Type `ls` and hit enter. You should see depot1 listed, as well as the root directories for other Rails projects you've done so far. If that's what you see, then you're in the right directory. Next, type the following:
   ```
   cp -r depot1 depot2
   ```
   `cp` means "copy". We're copying everything from the depot1 folder into the depot2 folder. The `-r` means "recursive", as in, copy not just what's in depot1, but everything in the subdirectories inside of depot1, and so on.

3. `cd depot2` now. This way you'll be working in depot2, and you can make changes to depot2 without having to worry about messing up your complete solution for the prior worksheet in depot1. Please be in the habit of making a new copy of your project every time you start a new worksheet!

4. Also, please be certain you use the same directory names I recommend in the worksheets (e.g. depot1, depot2). That way, it'll be easier for all of us to find what we need, because everyone's directory names will be consistent.

5. HTTP is stateless – it does not remember previous requests. This means we have to be careful if we want to remember an item that was added to the cart.
   a. In this worksheet, we will use sessions to remember states. A session consists of a hash and a session ID which is used to identify the hash itself.
   b. Every communication to or from the client's browser includes the session ID. This way, the server can keep track of all requests coming from the same browser.
   c. The `session` hash is pre-defined in Rails, similar to the `flash` hash. But there are several things we'll need to do to configure it for our use.
   d. Our depot application will store session data in a table in our database. We don't need to make this table manually – it's such a

common thing to do that it is supported by a gem. So edit the Gemfile to include the line: `gem 'activerecord-session_store'`

e. Next, enter `bundle install` at the command line so that the activerecord-session_store gem will be installed.

f. Then type
   `rails generate active_record:session_migration`
   to create a database migration for creating the session table.

g. The default is that the session is stored in a "cookie", which will severely limit its size. So we need to change the config/initializers/session_store.rb file. Comment out the existing line, and enter:
   ```
   Depot::Application.config.session_store
   :active_record_store
   ```
   (all on one line).
   Note that the application name "Depot" comes from the fact that in the first worksheet, we said "rails new depot", not "rails new depot1". So that's good – we'll keep the application name "Depot" even though it's now stored in the folder "depot2".

h. If your application server is running, you'll need to close and restart it after changing the above file.

i. Finally, `rake db:migrate` to apply the migration.

j. That's a lot of steps, but you don't have to memorize it. It's all listed in the Quick Reference file, in the "The Session" section, so know to look for it if you need it for another application sometime. Go review that section now.

k. At this point, we won't have to worry about any more details of storing sessions in the database – that will be handled for us. But we will need to manage the in-memory use of the `session` hash in a few controller actions, as we'll see.

6. Next, we will focus on constructing our cart. The cart holds data and business logic, so it's a model. However, we'll choose not to store a cart long-term in a database, so unlike other models we've made, we won't need a corresponding table for this model. So rather than doing something like `rails generate model Cart...`, we'll make a few files by hand.

   a. Note that while we won't be storing the cart long-term in a database, we will store it temporarily in the session – specifically, the `session` hash.

   b. Create a file in the app/models directory called cart_item.rb. In that file, define the class CartItem. It should have two fields: product (a Product object) and quantity. Apply attr_reader to both.

      i. You might want to take a quick look again at the Product class defined in a previous worksheet, to remind yourself of what is in it so you can use it in the remainder of this part.

   c. Write an initialize method that takes a Product object as a parameter, assigns it to the product attribute and sets the quantity attribute to 1.
   d. Write a method called increment_quantity that adds one to the quantity attribute.
   e. Write a method called title that returns the title of the product attribute.
   f. Write a method called price that returns the total price of the CartItem object – this is the price of the Product object times the quantity.
   g. Keep in mind the big picture here as we go: We have Products, defined in the previous worksheet. A CartItem is a Product and a quantity. It has methods to increment the quantity, get the title of the product, and get the total price (product price times quantity).

7. Let's define a Cart as an array of CartItem objects. Create a file in app/model called cart.rb. Now, let's define the Cart class.
   a. The Cart class should have one attribute: an array called items. Apply attr_reader to items.
   b. The initialize method doesn't have any parameters and initializes the items attribute to an empty array.
   c. Write a method called add_product. The method has one parameter: a Product object. The method looks for that Product object in the items array. If it finds it, it increments the quantity of that item by 1. Otherwise, it creates a CartItem object for that Product object with quantity 1 and adds it to the items array.
      i. You can use the << operator to append something to an array.
   d. Write a method called total_price. This method returns the total price of all the objects in the cart.
   e. Big picture: A Cart has a bunch of CartItems. When Products are added to the cart, we either increment the quantity of an existing CartItem, or add a new CartItem, as appropriate. The Cart can also compute the grand total for everything in the cart.

8. We are now done with the models that correspond to our cart. Let's write logic that will allow the cart to function. In the store controller, write a private method called find_cart. This method will return a Cart object for other methods to use.
   a. The cart will be stored in session[:cart]. So if this is nil, then the cart doesn't yet exist and should be created and stored in session[:cart].
   b. session[:cart] should then be returned.

9. Next, we need to write a method called add_to_cart in the store controller. This method will be called whenever a corresponding button is clicked.
   a. First, review the set_*resource* private method in any scaffold-generated controller for a reminder of the first time we saw the use of params[:id] to get an id from a URI.

   b. Add_to_cart should get the id of the Product added and add it to the current cart. To do this, first note that the method does not have any parameters. Rather, it should use params[:id] to get the id of the product from the URI.
   c. Find the corresponding product in the product database using Active Record. Then, use some of the methods you've already defined to get the cart and store it in @cart, and add the product to the cart.

10. In the index view for the store, modify the add_to_cart button you made in a previous worksheet. We need to make it call the add_to_cart action, as follows: <%= button_to 'Add to Cart', action:'add_to_cart', id:product.id %>.
    a. button_to is a Rails method that takes the button text and a hash as arguments.
    b. The hash we're passing has two key-value pairs: one for the action, and one for the id.
    c. The action determines what method will be called when the button is pressed. We're in the index view for the store, so the store controller's add_to_cart method will be called.
    d. The "product" is the parameter of the each loop in this view – it's the current product you're looking at, from the array of products.
    e. The add_to_cart method uses params[:id], so that's why we're providing an id here. There's nothing special about the name "id".
    f. Review the Quick Reference entry on button_to for an overview of all this.

11. Since the method add_to_cart is an action, when the button is pressed, Rails will execute the method and then look for a view that corresponds to that action. We need to create that view now, in the app/views/store directory. The view should display the content of the cart and the total price. (Use the method total_price of the Cart class.) Do not forget to add a route to this view in config/routes.rb; this should be a POST route, not a GET route, since we're changing something – the session, in this case. So use:
    post "store/add_to_cart".

12. To delete the old session data and start a new session do:
    rake db:sessions:clear.

13. In the store controller, implement an empty_cart method that sets the session[:cart] to nil, puts a notice about this in the flash, and redirects to the index page (use redirect_to action: "index").
    a. Be sure to display the flash notice in the index view, and add a route (post) for empty_cart!

14. Add a line at the end of your add_to_cart view to empty the cart; this should be a button similar to the add_to_cart button except that it should call the empty_cart method.

15. Test your application by going to http://.../store/index. Remind yourself of why this is the correct URI. Add a few books to the cart, including multiple copies of at least one. Verify that all prices are correct, taking quantities into account. Also verify that the Empty Cart button works.

So what's the big picture here? Follow through your code as you read the following important summary.

- Note in routes.rb that you have entries for the store index, add to cart, and empty cart. These are the only routes you'll need for the functionality as described below.
- You visit http://.../store/index.
- So the index action in store_controller.rb is called. It sets up the @products attribute.
- So the index.html.erb view is pulled up.
- There's nothing in the flash this first time, so nothing is displayed for that.
- The products set up in the index action are displayed in the index view.
- Suppose you then click the "Add to Cart" button. As you can see in index.html.erb, this calls the add_to_cart action in store_controller.rb, with id set to the current product's id.
- The add_to_cart action gets the product corresponding to the id sent by the button press. The cart is grabbed from the session, and the new product is added to the cart.
- Then the add_to_cart.html.erb view is called. The products in the cart are displayed, along with prices.
- Suppose you then click the "Empty Cart" button. So the empty_cart action is called (as you can see in add_to_cart.html.erb). The empty_cart action is defined in store_controller.rb.
- It sets the cart saved in the session to nil again, sets the flash, and redirects the index action.
- So the index action in store_controller.rb is called, which again sets up the @products attribute for use in index.html.erb, and the application continues.