Depot Application: Orders
CSC340

In this worksheet, we will focus on finishing the buyer's view. We'll enable the buyer to add items to a cart and check out.

1. Before you start, please go to the directory that contains depot2, and do the following:
   `cp -r depot2 depot3`
   Then `cd depot3` and work in there for this worksheet.

2. We need to generate a scaffold resource to store the orders. The name of the resource should be Order, and it should have the following fields: name (a string), address (text), email (string), and pay_type (string).

3. Apply the migration.

4. In the Order model, add some validations to ensure that all fields are present. (Check the Quick Reference if you need a reminder of how to do this.)

5. In the add_to_cart view, add a button called checkout that calls an action called 'checkout'.

6. In the store controller, write the action checkout. This action should:
   a. Find the cart and assign it to the @cart attribute.
   b. If the cart is empty, it should redirect the user to the index action with an appropriate flash[:notice] message.
   c. Otherwise, if the cart is not empty, it should create an Order instance field called @order. This will just be an empty object (uninitialized attributes) that we'll use for the form_for method in the next step.

7. Next, we need a view for our checkout action. The checkout view should have form elements where the user can enter their name, address, email, and pay_type. When the user clicks on the submit order button, the action save_order should be called. So the form_for command will look like the following:
   `<%= form_for @order, url: {controller:"store", action:"save_order"} do |f| %>`
   a. If you'd like a reminder for how forms work, look back at your solution for the "Using Forms" worksheet, where we worked with an Info model. But be sure you keep track of which open files are from that project, versus the one you're working on now! Otherwise things could get very confusing.
   b. Use text_field for every input except pay_type. For pay_type, use a drop box defined as follows:

<%= f.select :pay_type, options_for_select([["Credit Card", "CC"], ["Check", "C"], ["Money Order","MO"]])%>
options_for_select represents the options that will be listed in the drop box. The first parameter is what will appear in the drop box menu and the second parameter is the information that will be stored in the model.

8. Remember to add a post path for the checkout action in the routes.rb file.

9. In the store controller, implement the action save_order. It does not need a view; as you'll see below, it will just redirect the user to other views after doing some work.
    a. The save_order action needs to find the cart object and assign it to the @cart attribute.
    b. Capture the values from the form and put them in a new @order attribute using the params object:
        i. Note that the empty @order object we made in the checkout action is not available in save_order, due to the redirect implicit in pressing the "submit" button. The redirect generated a new HTTP request, and since HTTP is stateless, all in-memory objects were lost. But that's ok – we just needed that empty Order object for form_for anyway.
        ii. Also recall that since the checkout view uses form_for on @order, `params.require(:order).permit(:name, :address, :email, :pay_type)` will contain a hash of the attributes. So back in the save_order action, you can make a new Order object using that as the argument.
    c. Save the order to the database:
        i. If the save fails, redirect to the index with an appropriate flash message. Use: redirect_to :action => "index"
        ii. For now, if the save succeeds, clear the session[:cart] (`session[:cart] = nil`) since we won't need the cart contents anymore. Then redirect the user to the catalogue page (the index action) with an appropriate message.
    d. Don't forget the route!

10. Test your application to make sure it works properly. Be sure that your tests include verifying that valid orders are saved to the database, and incomplete orders (e.g. no address listed) don't get saved.

11. Let's generate a receipt for the buyer listing all the information about the order.
    a. Modify the save_order action:

i. First, we no longer want to redirect to the index action if the order saves correctly. Instead, redirect to a receipt action we're about to write.
ii. Second, we no longer need to clear the session[:cart], since we'll do that elsewhere in a moment.

b. Create the receipt action and the corresponding view. The view for the receipt action will need access to @order and @cart.
   i. Recall that @cart and @order were set in save_order. But then we needed to do a redirect_to – a new HTTP request – so those attributes were lost. So we'll have to get them again in the receipt action, to be used in the receipt view.
   ii. You can get the cart as usual using the find_cart method, which accesses the session.
   iii. To get the order, you might be tempted to first go to the save_order method, put the order in the session hash, and then extract it from the session in the receipt method. This would work for now, but would cause problems in later worksheets, when we'll make the Order object contain more data.
   iv. So instead of trying to save the entire order to the session in save_order, just save the order id. In receipt, you can then get the id from the session and use it to fetch the order from the database.
   v. We didn't try to do this with the cart, though, since recall we're not saving the cart in the database.
c. Once the @cart and @order variables are set up, clear out the session (set all values to nil) so that the user can start making a new order.
d. Do not forget to create a path for the receipt action in routes.rb.

12. Test your application.

13. Step through this all again as we've done before. Start by visiting a URI, observe what action is called, how it sets up the view, what view is called, what action is then called by pressing a button, etc.