Merchant

In this project you'll build an e-commerce site for a small grocery that wants to sell products directly to customers over the web. The project will be built in several discreet iterations.

NOTE

This tutorial is open source. If you notice errors, typos, or have questions/suggestions, please submit them to the project on GitHub [https://github.com/JumpstartLab/curriculum/blob/master/source/projects/merchant.markdown] .

Iteration 0: Up and Running

Part of the reason Ruby on Rails became popular quickly is that it takes a lot of the hard work off your hands, and that's especially true in starting up a project. Rails practices the idea of "sensible defaults" and tries to, with one command, create a working application ready for your customization.

Setting the Stage

First we need to make sure everything is setup and installed. See the Environment Setup [http://tutorials.jumpstartlab.com/topics/environment/environment.html] page for instructions on setting up and verifying your Ruby, Rails, and add-ons.

Generate the Project

Let's lay the groundwork for our project. In your terminal, switch to the directory where you'd like your project to be stored. I'll use `~/projects`.

Run the `rails -v` command and you should see your current Rails version. This tutorial was written with Rails **4.0.0**. Let's create a new Rails project:

Terminal

```
$ rails new merchant
```

Then `cd` into your project directory and open the project in your editor of choice, I'll use Sublime [http://www.sublimetext.com/2] .

Booting the Server

Start it up with this instruction:

Terminal

```
$ bin/rails server
```

Then try loading the address http://localhost:3000/ [http://localhost:3000] . You should see Rails' "Welcome Aboard" page. Click the "About your application's environment" link and it'll display the versions of all your installed components.

Scaffolding

Rails makes it really easy to begin modeling your data using scaffolding.

We'll start by thinking about a "product" in our store. What attributes does a product have? What type of data are each of those attributes? We don't need to think of EVERYTHING up front, here's a list to get us started:

- `title` which should be a `string`
- `price` which should be an `decimal`
- `description` which should be `text`
- `image_url` which should be a `string`

Why make `price` an `decimal`? If you make it a regular float, you're going to run into mathematic inconsistencies later on. Prices aren't real floats because the number of places after the decimal don't change — it's always two.

Ok, time to finally generate your scaffold. Enter this command:

Terminal

```
$ bin/rails generate scaffold Product title:string price:deci
```

Reading that line out loud would sound like "run the generator named `scaffold` and tell it to create an object named `Product` that has a `title` that is a `string`, a `price` that is a `decimal`, a `description` that is `text`, and a `image_url` that is a `string`"

The generator will then create about 30 files and directories for you based on this information.

We need to add extra options to our `price` column. Modify the `price` line in your freshly generated migration so it looks like this:

```
t.decimal :price, precision: 8, scale: 2
```

We give it two additional options: the `precision` controls how many digits the number can have in total. The `scale` controls how many digits come after the decimal. So this column will have a maximum value of 999999.99, which would be some expensive groceries.

Now you need to **run** this migration so it actually creates the `products` table in the database.

Terminal

```
$ bin/rake db:migrate RAILS_ENV=development
```

Technically, `RAILS_ENV=development` is redundant, because your rails environment already defaults to development.

You should see output explaining that it created the table named `products`.

Setting Up Strong Parameters

Rails defaults to a “whitelist” security system to protect application data from being mass-assigned. Mass assignment is what happens any time we set more than one attribute at once (for example, updating a Product’s title and price at the same time). There are certain fields that you don’t want to be mass-assignable for security reasons (e.g., we wouldn’t want an “admin” privilege to be mass-assignable on user, otherwise we might have security bugs where a user can elevate their own privileges).

The practical effect of this whitelist system is that we won’t be able to mass-assign any properties of any of the Products in our application until we remove that restriction. Since we used a generator to create the Product model we don’t need to worry about setting the accessible attributes ourselves, but just to confirm that they have been added, take a look at `app/controllers/products_controller.rb`, at the very bottom:

```
# Never trust parameters from the scary internet, only allow the whi
def product_params
  params.require(:product).permit(:title, :price, :description, :ima
end
```

This tells Rails to allow all four of our Product attributes to be mass-assign-able.

Creating Sample Products

Now, in your browser, go to http://localhost:3000/products [http://localhost:3000/products] .

You should get a page that says “Listing Products”, click the “New Product” link and it’ll bring up a very simple form. Enter the following data:

- Title: Green Grapes (1 bunch)
- Price: 2.00
- Description: 1 bunch of approximately 80 green grapes.
- Image_url: green_grapes.jpg

Click `Create`. If everything looks good on the next page, click the `Back` link. Click the “New Product” link and enter the second product:

- Title: Purple Grapes (1 bunch)
- Price: 2.25
- Description: 1 bunch of approximately 90 purple grapes.
- Image_url: purple_grapes.jpg

Create it and look at your products listing. Now you have a web store! (NOTE: The images will be missing for now, that's ok!)

How does Rails do that?

Let's take a peek at how this is all working. Browse your project files and look at the following files:

- `config/routes.rb` When you request a page from the application this is the first place Rails goes. See the line `resources :products`? Resources are things that follow the RESTful web conventions. For instance, if the router receives a GET request for `/products` it should look in the `app/controllers/products_controller.rb` for a method named `index`. After finding the `products` entry in the Routes file, Rails knows to come to this file and run the `index` method. You'll see the `index` method like the code below. The second line is the most important — it reads "Find all the Products and stick them in the variable `@products`".

```
def index
  @products = Product.all
end
```

- Now take a look at `app/views/products/index.html.erb` This is the view template which is responsible for creating the HTML sent to the browser. The template is written in a format named **ERB** which allows us to mix HTML and Ruby. The first few lines of this files just look like plain HTML. On line 10, though, you see some ERB syntax. Look at the example below. See the `<%` and `%>` in line 17? Those tags mark the beginning and end of Ruby code inside the ERB file. Basically they mean "evaluate whatever code is between these markers as though it were part of a Ruby program." Then in line 19 you see a variation where the opening tag has an equals sign like this: `<%=`. When there is **no** equals sign, the code between the markers is run **silently**. When the beginning marker has the *equals sign*, erb will **output** the result of the code into the HTML.

```
 1  <h1>Listing products</h1>
 2
 3  <table>
 4    <thead>
 5      <tr>
 6        <th>Title</th>
 7        <th>Price</th>
 8        <th>Description</th>
 9        <th>Image url</th>
10        <th></th>
11        <th></th>
12        <th></th>
13      </tr>
14    </thead>
15
16    <tbody>
17      <% @products.each do |product| %>
18        <tr>
19          <td><%= product.title %></td>
20          <td><%= product.price %></td>
```

```
        <td><%= product.description %></td>
        <td><%= product.image_url %></td>
        <td><%= link_to 'Show', product %></td>
        <td><%= link_to 'Edit', edit_product_path(product) %></td>
        <td><%= link_to 'Destroy', product, method: :delete, data: {
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Product', new_product_path %>
```

Now you've got a database-driven web application running and Iteration 0 is complete.

Iteration 1: Basic Product Listings

So now you might be impressed with yourself. You've got a web store just about done, right? Real stores have a ton of **information** …pictures, data, reviews, categories not to mention the ability to actually **buy** something. We'll get there — for now let's make our store look a little more respectable.

Working with Layouts

In the first iteration I lied to you about how the view step works. When you're looking at the products listing, Rails isn't just grabbing the `index.html.erb`. It's also looking in the `app/views/layouts/` folder for a **layout** file. Expand that directory in your editor and you'll see that the scaffold generator created a file named `application.html.erb` for us.

You can think of a layout like a wrapper that is put around each of your view templates. Layouts allow us to put all the common HTML in one place and have it used by all our view templates. For instance, it'd be a pain to have to write out the whole `<head>` section in each of our ERB templates. Open the layout file and you'll see a lot of the HTML framework is already in the layout file.

If you're on Amazon each of their pages has a certain look and feel. They have common navigation, stylesheets, titling, etc. The layout is where we take care of these common elements. In the `<head>` section of the layout it currently has line with a `<title>` — change it so it reads like this:

```
<title>FoodWorks - Products: <%= controller.action_name %></title>
```

Hit save, switch back to your browser with the product listing, and hit refresh. The title of your browser tab should now say "FoodWorks – Products: index". Then click the `New product` link. The window title should now say "FoodWorks – Products: new". Even though the `index` and `new` actions have different view templates they share the same layout, so the change we made shows up in both places. Let's add a little more to the layout file…

- Download that stylesheet (styles.css [http://tutorials.jumpstartlab.com/assets/merchant/styles.css]) and put it into your project's `app/assets/stylesheets/` folder.
- Next let's add a little structure to our pages to make CSS styling easier. Check out your application layout at `app/views/layout/application.html.erb` and modify everything from `<body>` to `</body>` so it matches the code below:

```
<body>
<div class="wrapper">
    <div class="header">
      <h1>FoodWorks</h1>
      <p>Your Online Grocery</p>
    </div>

    <p style="color: green"><%= flash[:notice] %></p>

    <div class="sidebar">
      (sidebar)
    </div>

    <div class="main">
      <%= yield %>
    </div>

    <div class="footer">
      FoodWorks Online Grocery<br/>
      A Rails Jumpstart Project
    </div>
</div>
</body>
```

Now save your layout and refresh the products listing. It should look a little prettier, but we haven't changed much about the content yet.

Editing a View Template

Let's open the `app/views/products/index.html.erb` view template so we can make some changes.

- Change the title to H1 tags with the text "All Products"
- The link at the bottom is currently "New product" — let's change it to "Create a New Product" like this:

```
<p><%= link_to "Create a New Product", new_product_path, id: "new_pr
```

- Then let's restructure the table. See the line that says `<% @products.each do |product| %>`? That means "for each of the things in the variable `@products`, take them one at a time, call them `product`, then do everything in between this line and the one that reads `<% end %>`" So for each `product`, the existing template creates one TD for the `title`, one for the `price`, one for the `description` and so on. Let's simplify the products like this:

```
<% @products.each do |product| %>
  <tr>
```

```
    <td><%= image_tag "products/#{product.image_url}" %></td>
    <td><span class="product_title"><%= product.title %></span><%= p
    <td><%= product.price %></td>
  </tr>
<% end %>
```

Try looking at the page in your web browser. Rework the table's THs to match up with the new TDs.

Download the sample images for products here: products.zip [http://tutorials.jumpstartlab.com/assets/merchant/products.zip] . Move the images into `app/assets/images/products/`.

Save that and check out your products index at http://localhost:3000/products [http://localhost:3000/products] .

Fixing the Price Display

When you look at the products index the price is a little ambiguous. It doesn't have a currency marker. Let's clean it up with a view helper.

A helper, in Rails, is code that helps you manipulate data as part of the presentation. We want to make a helper where we can send in a number like "2.25" and it gives us back "$2.25", the format our shoppers are anticipating.

- Open the file `app/helpers/products_helper.rb`
- Between the line that starts with `module` and the `end`, add this method:

```
def print_price(price)
  "$#{price}"
end
```

Go back to your `index.html.erb` file and change `<%= product.price %>` to `<%= print_price(product.price) %>`. Save it and refresh your products display. Notice anything?

If you have a price that ends in a zero, like your Green Grapes at $2.00, you'll see that the trailing zeros are being cutoff. That's no good.

Look at your `print_price` helper again. What's the right fix? One approach is to use the `format` method in Ruby like this:

```
def print_price(price)
  format("$%.2f", price)
end
```

That works great, check it out in your browser. The only objection is that it doesn't internationalize. When we want to start selling groceries in the EU, we'll need to rewrite this code.

Rails has a built-in helper method for exactly this purpose named `number_to_currency` . Its full description is here: [http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html#method-i-number_to_currency] http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html\#method-i-number_to_currency [http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html\#method-i-number_to_currency]

We can use it just like this:

```ruby
def print_price(price)
  number_to_currency price
end
```

Check out the results in your browser; your store is starting to look good!

Linking to the Product Page

When you're on the index page there's no way to get to the "show" page for an individual product. Go back to the `index.html.erb` and use the `link_to` helper to insert links to the show page. `link_to` requires two parameters: what the link should **say** and where it should **point**. To trigger a show action, point the link towards `product_path(product)`.

Working on the New Product Page

Click the `Create a New Product` link on your index page. Rails will access your `products_controller`, find the `new` method, run that code, then load the `new.html.erb` view template. Open that ERB file and you'll see there isn't much there.

Change the title to an H1 that reads "Create a New Product." Then to manipulate the form itself we'll need to jump into `_form.html.erb`.

- Look at the line `<%= f.label :price %><br />`. This tells Rails to create an HTML `label` for the thing named `:price` and, by default, it puts the text `Price` there. We want it to say "Price (like 2.99)" instead. We add in the desired text as a second parameter like this: `<%= f.label :price, "Price (like 2.99)" %><br />`
- Save your ERB and reload your form to make sure everything is looking ok

Create a new product and enter in this information EXACTLY:

- Title: Oranges
- Price: $2.99
- Description: Bag of 6 Valencia oranges.
- Image_url: oranges.jpg

Click `Create` then look at the page you get back.

Validating the Price

What's up with the price? Is that what you expected?

We tried to put "$2.99" to the database, but it's expecting a number, not something with a dollar sign and a period. How does it end up with $0?

The data is coming in from our form as a string, then Rails is trying to coerce it into a decimal. Open `bin/rails console` and try calling `"$2.99".to_f` to convert it to a float. This is similar to the coercion that happens when we convert it to a decimal. We get zero!

How can we help the user not make this mistake? Adding a validation.

- Open the file `app/models/product.rb`
- Before the class' `end`, add this validation: `validates_numericality_of :price`
- Get back to your product listing and `destroy` the oranges that we just created
- Click `Create a New Product` and enter the orange information just as we did above and click `Create`

Now you should see some great things that Rails does for "free". It keeps us on the product creation screen, it tells us that there was a problem, it tells us what the problem is, then it highlights both the label and the form field in red. Makes it pretty clear, right? Take off the dollar sign so your price just says "2.99" and click `Create` again.

This is an opportunity to make things easier on our users. Typing a dollar sign in the price is perfectly reasonable from their perspective. We can handle it transparently on the server side.

When the form is being processed it passed the hash of values from the form to `Product.new`. ActiveRecord then goes through the values in that hash and stores them into the new record. It doesn't directly manipulate the object's attributes, it uses setter methods kind of like this:

```ruby
def price=(input)
  self.price = input
end
```

We don't see that method in our code because ActiveRecord generates it on the fly at load time. But we can override the method if we want to do something special, like delete a dollar sign:

```ruby
def price=(input)
  input.delete!("$")
  super
end
```

The first line does an in-place delete (`delete!` as opposed to `delete`), removing any dollar signs. Then, with the value changed, `super` invokes the original `price=` method.

Try it out in your console and browser. Set prices like “2.99” and “$3.99” — they should both work as expected.

Now, go ahead and create a few more sample products. Look at the folder of images that you copied over to see what’s available. Make at least 5 products for your store.

Iteration 1 is complete!

Iteration 2: Handling Stock

Any good store needs to manage stock. When customers are shopping they should be able to see the current stock. When people buy something, the stock goes down. Administrators should be able to arbitrarily change the stock count.

Modifying the Database

Anytime we’re tracking new data we’ll need to modify the database. Jump over to your Terminal and generate a migration with this command:

Terminal

```
$ bin/rails generate migration add_stock_to_products
```

After it generates, open the migration (look in `db/migrate`) and in the `change` add the line below.

```
add_column :products, :stock, :integer, default: 0
```

Run the migration with `bin/rake db:migrate` then the column exists in your database.

Adding to the Products Listing

Let’s open up the view for our products index (`app/views/products/index.html.erb`) and add in a column after `Price` for `Stock` in the THs. Down in the TDs, write this:

```
<td><%= print_stock(product.stock) %></td>
```

So that’s expecting to use the helper method named `print_stock`. Here’s the logic we want to implement:

- If the product is in stock, print the following where ## is the number in stock:
  - `<span class="in_stock">In Stock (##)</span>`
- If it’s out of stock, print the following:
  - `<span class="out_stock">Out of Stock</span>`

Go into the `products_helper.rb` and create a method named `print_stock` then fill in the blank lines with the stock messages:

```ruby
def print_stock(stock)
  if stock > 0

  else

  end
end
```

Hint: check out the `content_tag` method (http://api.rubyonrails.org/classes/ActionView/Helpers/TagHelper.html#method-i-content_tag [http://api.rubyonrails.org/classes/ActionView/Helpers/TagHelper.html#method-i-content_tag])

With the helper implemented refresh your products index and you should see all products out of stock.

Making the Stock Editable

Hop into the show page for your first project then click the `Edit` link.

This edit form only shows the original fields that were there when we ran the scaffold generation, but it's easy to add in our stock. Open the form template at `app/views/products/_form.html.erb`

Using the title and price as examples, write a paragraph for the stock including the label and a text field. Refresh your web browser and you should see the stock field available for editing.

Since it's just a raw text field, let's add some validation to the `product.rb` model to make sure we don't put something crazy in for the stock. Check out the Rails Guide on Validations [http://guides.rubyonrails.org/active_record_validations.html] and figure out how to write ONE validation that makes sure:

- stock is a number
- stock is an integer
- stock is greater than or equal to zero

Once your validation is implemented, give it a try with illegal values for `Stock` like `-50`, `hello!`, and `5.5`. Is it failing?

Dealing with Strong Parameters

Your validation probably isn't working, right? Your form isn't changing the value of stock, but it also isn't showing a validation error. What's up?

Take a look in the `ProductsController` again. Unless you anticipated this problem and fixed it already, `stock` will not be in your `product_params` permitted list, and that would stop you from being able to submit an adjustment to `stock` via the form.

Add `stock` to the `permit` list and retry your good and bad stock values.

```
def product_params
  params.require(:product).permit(:title, :price, :description, :ima
end
```

With those validations implemented, add stock to most of your products so we can do some shopping.

Iteration 2 is complete!

Iteration 3: Designing Orders

We've got products, but until we have a way to create orders then our store isn't going to make any money.

Brainstorming the Data Structure

Let's think about what an "order" is:

- It's a collection of products to purchase
  - Each of those has a quantity and a unit price
- It has a total price
- It has a customer
- It has a status like "unsubmitted", "needs_payment", "needs_packing", "needs_shipping", and "shipped"

Looking at this from the database perspective, we'll need two objects:

- `OrderItem` One piece of an order, this will be a single product ID and a quantity of how many are being ordered
  - `product_id`: (integer) the `Product` that this `OrderItem` is referencing
  - `order_id`: (integer) the `Order` that this `OrderItem` is referencing
  - `quantity`: (integer) the quantity of this product desired
- `Order` It will have many `OrderItem` objects, belong to a customer ID, and have a status code
  - `user_id`: (integer) the user that this order belongs to
  - `status`: (string) the current status of the order

With that in mind, go to your Terminal and generate some scaffolding and migrate the database:

Terminal

```
$ bin/rails generate scaffold OrderItem product_id:integer or
  bin/rails generate scaffold Order user_id:integer status:st
$ bin/rake db:migrate
$
```

The Data Models

From there we need to build up our two data models and express their relationships.

The `OrderItem` Model

Jump into `app/models/order_item.rb` and we'll work on relationships. Since the `OrderItem` holds a foreign key referencing an `Order`, we'd say that that it `belongs_to` the `Order`. Add the relationship to the model like this:

```
class OrderItem < ActiveRecord::Base
  belongs_to :order
end
```

There's also a relationship between an `OrderItem` and a `Product`. A `Product` is going to be ordered many times, we hope, but an `OrderItem` is only going to connect with a single `Product`. This is a "One-to-Many" relationship where one `Product` is going to have many `OrderItems` and an `OrderItem` is going to `belong_to` a product.

Add the second `belongs_to` relationship in `order_item.rb`

Then, add a validation to make sure that no `OrderItem` gets created without an `order_id` and a `product_id`.

The `Order` Model

Next, let's turn our attention to `app/models/order.rb`. We said previously that an `OrderItem` would `belong_to` an `Order`. We definitely want people ordering as many products as they want, so an `Order` should have many `OrderItems`. Add it like this:

```
class Order < ActiveRecord::Base
  has_many :order_items
end
```

Products to Order Items

We won't actually need it for our application, but as long as a `Product` is related to an `OrderItem`, you should go ahead and add the `has_many` association in `product.rb`.

Implementing an Order Workflow

That was the easy part. Now it's going to get tricky.

Add to Cart Links

Let's open the `index` view for our `products` (`app/views/products/index.html.erb`). How do we want the shopping process to work? Each product description should have an "Add to Cart" link that adds that item to the customer's cart. Once they click that the customer should be taken to their order screen where they can set quantities and checkout.

First we'll create the "Add to Cart" link. In the `<th>` lines near the top of the index add `<th>Buy</th>`. Then in the TDs, underneath the stock line, let's add a new cell:

```
<td><%= link_to "Add to Cart", new_order_item_path(product_id: produ
```

This says "create a link with the text 'Add to Cart' which links to the new `order_item` path and sends in a parameter named `product.id` with the value of the ID for this `product`.

Reload the index in your browser and you new link should show up.

Tracking an Order

When you click the "Add to Cart" link you see a plain scaffolding form prompting the user for the product id, order id, and quantity. That's not acceptable.

When the user clicks "Add to Cart", the system should:

- Find or create a new `Order` for the current visitor
- Add the item to the `Order` with a quantity 1

The first part is tricky because HTTP is a stateless protocol. There isn't any continuity between requests, so it's hard to identify a user. But in situations like this one, we need to keep track of users across many page clicks.

We'll take advantage of Rails' session management here. The session will allow us to track data about a user across multiple requests.

Managing Orders in a Before Filter

Open your `app/controllers/order_items_controller.rb` file. This controller is what handles the "operations" of a web request. The request starts at the router (the `config/routes.rb` file), gets sent to the correct controller, then the controller interacts with the models and views. Think of the controller as the "coach" — it calls all the shots. At the top of the controller file, just below the `class` line, add this code:

```
before_action :load_order, only: [:create]
```

We're declaring a `before_action`. This tells Rails "before every request to this controller, run the method named `load_order`". The `only: [:create]` part tells rails only to run this method before calls to the create action. If we didn't include the `only: [:create]` parameter then the `load_order` method would be called before every action in the `order_items` controller.

If you think about it, it doesn't really make sense for us to call the `load_order` method before every action; we only need to call it before we add a new order item.

We need to define the method named `load_order`. At the bottom of the same file, below the line that says `private` add the following:

```
def load_order
  begin
    @order = Order.find(session[:order_id])
  rescue ActiveRecord::RecordNotFound
    @order = Order.create(status: "unsubmitted")
    session[:order_id] = @order.id
  end
end
```

Rails provides us access to the user session through the `session` hash. This method tries to find the `Order` with the `:order_id` in the session and stores it into the variable named `@order`. If the `session` hash does not have a key named `:order_id` or the order has been destroyed, Rails will raise an `ActiveRecord::RecordNotFound` error. The `rescue` statement watches for this error and, if it occurs, creates a new `Order`, stores it into the variable `@order`, and saves the ID number into `session[:order_id]`.

With that code in place refresh your Products index and… nothing looks different.

That's because nothing happened. The `load_order` method gets called when we hit `POST /products`, but we've been hitting `GET /products/new`.

We could change the link so that it tricks Rails into accepting a POST, but it's generally cleaner to just create a form. That way search spiders and other internet bots won't create carts willy-nilly.

Replace the *Add to Cart* link with this:

```
<td><%= button_to "Add to Cart", order_items_path(product_id: produc
```

Reload the index page, and click the *Add to Cart* button. The page blows up, complaining about not finding a parameter:

```
param not found: order_item
```

That's because the form that gets created by the `button_to` helper doesn't nest data in an `order_item` hash, it simply submits to the URL provided.

We need to update the `create` action in the OrderItemsController. Change this:

```
def create
  @order_item = OrderItem.new(order_item_params)
  # ...
end
```

to this:

```
def create
  @order_item = OrderItem.new(product_id: params[:product_id])
  # ...
end
```

If you refresh the index page and click *Add to Cart* you'll end up on the new page for the order item [http://localhost:3000/order_items/new] with a complaint that you're missing the order_id.

We have the order stored in an instance variable. Let's use that.

```
def create
  @order_item = OrderItem.new(product_id: params[:product_id], order_
  # ...
end
```

Now if you click the add to cart button, you will end up on the Order Item's show page.

It would be more helpful to go to the order page. In the `create` action of the order items controller change the response for HTML to redirect to the order:

```
format.html { redirect_to @order, notice: 'Successfully added produc
```

Try adding a product to your cart again.

Improving `load_order`

In general we should avoid using exceptions for control flow. Our implementation of `load_order` is expecting an exception to be raised under a somewhat normal circumstance: a new user arrives on the site. We should redesign this method to not rely on `begin`/`rescue`/`end`.

`ActiveRecord` provides us with dynamic finders and initializers for all our models. For a `Product`, we can do any of the following:

```
Product.find 2
Product.find_by_title "Green Grapes"
Product.find_by_image_url "purple_grapes.jpg"
```

We can call a class method `find_by_` then give it the name of any model attribute to search for a match. We can also use `find_all_by_` to fetch an array of all matches.

How does that help us here? Well, Rails takes it a step further:

```
Product.find_or_initialize_by_id(5)
Product.find_or_initialize_by_title "Apples"
```

The `find_or_initialize_by_` method will first attempt to find a match in the database for the specified attribute and value. If it finds one, that object will be returned. If it **doesn't** find a match, it will build one in memory and return the new object.

This object will not yet have been saved to the database. If we wanted to build it and save it in one step, we could use `find_or_create_by`. Returning to our `load_order` method, we can implement the technique like this:

```ruby
def load_order
  @order = Order.find_or_initialize_by_id(session[:order_id], status
  if @order.new_record?
    @order.save!
    session[:order_id] = @order.id
  end
end
```

Note that the `status: "unsubmitted"` will only set the `status` if it is initializing a new object, it will not change the value of records found in the database.

From there we look to see if it is a `new_record?`, which is true when the record has not yet been stored to the DB. If it is new, save it to the DB so it gets an ID, the store that ID into the user's `session`.

Cleaning up

We don't use the new form. Let's get rid of the `new` action in the controller. While we're cutting code, we won't need the `show` or `index` actions either, so delete them.

Rewriting the Create Action

Look at the server log for the last request. You should see "Started POST" then look at the parameters line. See the `product_id` in there? We have all the information we need to rewrite the `create` action. The `create` currently looks like this:

```ruby
def create
  @order_item = OrderItem.new(product_id: params[:product_id], order_

  respond_to do |format|
    if @order_item.save
      format.html { redirect_to @order, notice: 'Successfully added
      format.json { render action: 'show', status: :created, locatio
    else
      format.html { render action: 'new' }
      format.json { render json: @order_item.errors, status: :unproc
    end
  end
end
```

Building the `OrderItem`

We can build the `order_item` through the relationship with the order like this:

```ruby
def create
  @order_item = @order.order_items.new(quantity: 1, product_id: para
  # ...
end
```

The `order_id` will be set by the relationship, then we explicitly set the `quantity` to one and the `product_id` comes from the request parameters.

Go back to your products `index` page, click an "Add to Cart" button.

Displaying an Order

You should now be redirected to http://localhost:3000/orders/1 [http://localhost:3000/orders/1] which is your current order. You're looking at the order, you think some products have been added, but we're not doing anything to display them yet.

Add this snippet somewhere on the page:

```
<p>
  Item Count: <%= @order.order_items.count %>
</p>
```

Reload the page and you should see the **Item Count** display 1. Go back to your products listing and click "Add to Cart" to add more items. You should see this counter increasing each time.

Our shopping experience has a long way to go, but it's getting there.

Iteration 3 is complete!

Iteration 4: Improving the Orders Interface

We'll continue working on our `app/views/orders/show.html.erb` so open it in your editor and load an order in your web browser.

Reworking the Order's `show` View

Replace the code in the `show.html.erb` with this:

```
<h1>Your Order</h1>

<table>
  <tr>
    <th>Customer</th>
    <td><%= @order.user_id %></td>
  </tr>
  <tr>
    <th>Status:</th>
    <td><%= @order.status %></td>
  </tr>
  <tr>
    <th>Items:</th>
    <td><%= @order.order_items.count %></td>
  </tr>
  <tr>
    <th>Items</th>
    <th>Title</th>
    <th>Quantity</th>
    <th>Unit Price</th>
    <th>Subtotal</th>
  </tr>
  <!-- more code will go here -->
</table>
```

Save that then refresh your order view. The basics are there, but it still isn't showing what items are in the order.

Displaying Individual Items in an Order

Remember when we declared that an `Order` `has_many` `OrderItems`? We did that so we could easily access an order's items. Remove the line that says `<!-- more code will go here -->` and replace it with this:

```
<% @order.order_items.each do |item| %>

<% end %>
```

Inside those lines put the HTML and data you want rendered for `each` item in the `@order`. Follow the headings that already exist. You'll need to create a new `TR` that contains...

- A blank TD
- The `item.product.title` in a TD
- The `item.quantity` in a TD
- The `item.product.price` in a TD and use the `print_price` helper
- A blank TD, temporarily, for the subtotal

Refresh your view and you should see your individual `OrderItems` listed out.

Displaying Product Images

Integrate images of the products into your order display into the first TD of the row. As you did in the products listing, the image can be inserted by using Rails' `image_tag` helper method along with the path to the image (which the `Product` model stores). Try and figure this out on your own, but if necessary go look at your `app/views/products/index.html.erb` file.

Order Calculations

We need to both calculate the subtotals for individual items and the total cost for the entire order.

`OrderItem` Subtotal

The subtotal is the easier part because it only involves one object, the `OrderItem`. We can ask an `OrderItem` to output its subtotal like this:

```
<%= print_price item.subtotal %>
```

If you refresh that will crash because it's expecting `item` to have a method named `subtotal`. Open the `OrderItem` model and define a method named `subtotal` that returns the quantity times the product's price.

When you think you've got it, refresh the view and check out your results. It's not very convincing since our quantities are all 1 right now, but it's a start.

Calculating the Order Total

Now we're displaying the individual items and their costs, but we don't have a total for the order. Let's open the `Order` model (`app/models/order.rb`) and create a method named `total`. Our `OrderItem` model already implements a `subtotal` method that gives us the total for that single item, so what we need to do is add up the `subtotal` from each of the `order_items` in this order.

I'm going to leave that up to you.

With that method written, go back to your order's `show` view and add another line at the bottom with a TH that says "Order Total" and a TD that displays the total. Remember to use your `print_price` helper method to get the right formatting.

Manipulating the Order

We can add items to the order, but we can't remove them.

Removing a Single Item from the Order

Let's add a "remove" link for each item in the order.

Removing a single item from the order is actually pretty easy. To get a hint, let's look at some scaffolding views that we aren't actually using. Open up `app/views/order_items/index.html.erb`. See how it makes the "Destroy" link? Let's grab that whole line.

Move back to your order's `show` template go to the area where you're displaying the individual order items. Modify your table to make an appropriate space for a "remove" link and paste in the code we got from the other template. You can change the word "Destroy" to "Remove" be less dramatic. Also change `order_item` to `item`, since that's what we called it in that view's `each` block.

Click one of the delete links and it almost works. The `destroy` action in `OrderItemsController` is being triggered, but after destroying the object it redirects to the `index` of `OrderItemsController`. Instead, make it redirect to the order, then go back and try deleting another item.

Clearing All Items from an Order

Deleting a single item was easy, but deleting all of them? We have two options:

1. Create a custom `empty` action in `OrdersController` and have it loop through destroying the individual `OrderItems`
2. Just destroy the Order

I like easy, so let's pick #2!

On the `show` view for your `Order`, add a link like this:

```
<%= link_to "Empty Cart", @order, data: { confirm: 'Are you sure?' }
```

Hop into the `destroy` action of `OrdersController` and change the redirect so it sends you back to `products_path`.

Then give it a try.

Destroying `OrderItem` Records

If you go to "http://localhost:3000/order_items/" you'll see all the `OrderItem` records stored in the database. Go to your console and erase all the existing `Order` objects:

IRB

```
2.1.1 :001> Order.destroy_all
```

Refresh the `OrderItem` listing and… they're all still there? If an `Order` gets destroyed then we want the `OrderItem` objects to go too!

Go into `Order` model. Change the `has_many :order_items` line to this:

```
has_many :order_items, dependent: :destroy
```

Now, when an `Order` is destroyed all the associated order items will get destroyed too. This does **not** remove the `OrderItem` objects that are already orphaned in the database. Kill those through the console:

IRB

```
2.1.1 :001> OrderItem.destroy_all
```

Then, through the web interface, add a few items to a new `Order`. Destroy that `Order`, and check that the associated items are gone too. When it works, Iteration 4 is complete!

Iteration 5: Dealing with Order Quantities

Our order screen is getting powerful, but there are still some features we should add around managing quantities.

No Double-Items

I'm sure you noticed that new order items are getting added to the cart each time the user click "Add to Cart". We don't want two listings for Green Grapes, we want one `order_item` with a quantity of two. No problem!

This is how it should work:

- When the user clicks the Add to Cart button…
    - If that product is already in the order, increase the quantity by one
    - If it's not in the order, add it to the order with quantity one

Look at the `create` method in your `order_items_controller`. The first line is always creating a new `OrderItem`, but we want to look for one with the matching `product_id` first. We could do this ourselves with and `if` statement, but let's take advantage of Rails' `find_or_initialize_by` like this:

```
@order_item = @order.order_items.find_or_initialize_by_product_id(pa
```

We can give `find_or_initialize_by` any attribute name of the object, in this case `product_id`, and pass in the `product_id` we're looking for. If there is a matching `OrderItem` already attached to this `Order` with this `product_id`, it'll find it and give it back to us. If it doesn't exist, one will be created.

Try using this in your browser. As you add products to your cart you should see that items to not get repeated, but the quantity is not yet going up when we add the same product twice.

Empty your cart, and add a new item. Now it blows up because the item doesn't have a quantity at all.

Incrementing Repeated Items

If the finder is creating a new `OrderItem`, we want to set the quantity to 1. If it finds an existing `OrderItem`, we want to just increment it by one. We can do both of these in one instruction **if** the default `OrderItem` quantity is zero.

We set the default value in the database, and to change the database we'll need a migration. From your command line:

Terminal

```
$ bin/rails generate migration add_default_quantity_to_order_
```

Then open that migration and in the `change` method add this line:

```
change_column :order_items, :quantity, :integer, default: 0
```

Run the migration with `bin/rake db:migrate`. If you'd like to see the results, create a new `OrderItem` from your console and you'll see it starts with the quantity 0.

Now that the default value is set, your `create` action in `OrderItemsController` is easy. If it's incrementing an existing item, then we add one to the existing value. If it's a new record, we want to increment the counter from zero to one. Since zero is the existing value of a new record, we can simply add one to the quantity and not care if it's a new record or an existing one.

Add a line to your `create` action that adds one to the order item's quantity before the `.save` is called.

Try adding an item to your cart multiple times and you should see the quantities, subtotal, and total moving up!

Managing Quantities in the Order

We never made a way for customers to easily modify the quantity of each item in the order. Let's implement that now. There are several ways we could do this from an interface perspective. We'll implement an easy one.

Look at the `show` template for the `order`. Find the TD where we currently just print out the quantity for that item. Let's change it to a link like this:

```
<td><%= link_to item.quantity, edit_order_item_path(item) %></td>
```

That says "create a link with the text of the link displaying the item's quantity and the link pointing to the `edit_order_item` action and tell it that we want to edit the item named `item`".

Save and refresh your browser. Click one of the resulting links under quantity.

Cleaning Up the Order Items Edit

You are back to some ugly scaffolding code and this form is way too flexible. We don't want people changing the product ID or the order ID, just the quantity. Open the `form` partial for `order_item` (`app/views/order_items/_form.html.erb`) and make the following changes:

- Remove the `label` and `text_field` for `product_id` and, instead, just print out the product's title
- Remove the whole section for the `order_id`. It doesn't need to be displayed or editable.

Refresh your page and make sure the form is displaying properly. Enter the quantity desired as `5` then click Update.

It worked, kind of. It saved the new quantity, but the controller tries to bounce you to the `show` template for `order_item`. What we'd really like is to bounce back to the `show` for the `order`. Open up `app/controllers/order_items_controller.rb`. Scroll down to the `update` method, and change the `redirect_to` so it points to the order.

Go through and try changing the quantity again and you should return to the order with the updated quantities.

Remove Items with 0 Quantity

Sometimes instead of clicking "remove" to remove an item from an order, users will set the quantity to zero. Try doing this now with one of your existing orders. What happens?

To tell you the truth, I expected an error.

We put in a validation that `order_item` couldn't have a zero or negative quantity because that wouldn't make any sense — right? Right? Nope, missed it.

Open up your `app/models/order_item.rb` and add a validation that ensures `quantity` is a number, an integer, and greater than zero.

Now go back to your order screen, edit an item's quantity to zero, then click update. You should get an error message sending you back to the edit form saying that `quantity` must be greater than zero. This is good for our data integrity, but ugly for our users.

Now look at `app/controllers/order_items_controller.rb` and specifically the `update` method. We want to short-circuit this process if the incoming `params[:order_item][:quantity]` is zero.

Restructure the logic with a conditional statement like this:

- find the `order_item` by the `id`
- if `params[:order_item][:quantity].to_i` is equal to zero
    - destroy the `order_item`
    - redirect to the order and set the `flash` to say that the item was removed
- elsif the attributes successfully update…
    - redirect to the `Order` with a notice `"Successfully updated the order item"`
- else
    - display the edit form again

Now that we're finding the order item in the update action, we no longer want to run the `before_action`:

Remove `:update` from the list:

```
before_action :set_order_item, only: [:show, :edit, :destroy]
```

Now if you update an items quantity to zero you shouldn't get an error and the item will be removed.

Test it out and confirm that setting the quantity to zero removes an item from the order.

More Intelligent Stock Checking

Now that we can make all these changes to the quantity being ordered it makes our current stock checking ineffective. If there's at least one of an item in stock, our app will say it's "in stock". But if the customer is trying to order more than the current stock, we should change that notification.

Displaying Stock on the Order Page

Let's start by opening the `app/helpers/products_helper.rb` file and finding the `print_stock` method we created earlier. We want to create logic like this:

- if there are none of the items in stock
  - return the "out of stock" line
- else if there is enough stock to fulfill the requested number
  - return the "in stock" line
- else if there is some stock, but not enough to fulfill the requested number
  - return this: `content_tag(:span, "Insufficient stock (#{stock})", class: "low_stock")`

But how will the helper know what quantity the current order is requesting? We'll have to add a second parameter. So change

```
def print_stock(stock)
```

to...

```
def print_stock(stock, requested)
```

Then use the `requested` variable to implement the logic above.

Go back to your your `show` view template for `orders` (`app/views/orders/show.html.erb`). Try to add in a column to the display that prints out the stock status of each of the items in the order. Here are the steps you need to do:

- Add the header for `Stock` between `Quantity` and `Unit Price`
- Add the TD that calls the helper method `print_stock` and passes in the value of this item's product's stock and the quantity requested from this `order_item`

Refresh your browser and confirm that it's working. Try setting your quantities to trigger the "Insufficient Stock" message.

Displaying Stock on the Products Index

Now go back to your Products listing page. Problem? Your products listing is also trying to use that `print_stock` helper, but it's just sending in one parameter. Now we're getting the error message that the method is expecting two parameters. How to fix it? There are two ways.

The ugly way would be to open the products `index` view and change our call to the helper, adding in a `0` for the number requested. That'd work, but we don't like ugly. If we end up using the helper anywhere else, we'd have to remember to always put in this hack.

Instead we'll improve the helper, so switch back to that file. Ruby has a great way of implementing optional parameters. We can set it up so calling the helper with one parameter will print whether or not the item is in stock, and sending in two parameters will check if there's sufficient stock. All we need to do is change...

```
def print_stock(stock, requested)
```

to...

```
def print_stock(stock, requested = 1)
```

With that change, if we send in a value for `requested` the method will use it. If we don't send in a value for `requested`, and thus have only one parameter, it'll just set requested to one. This will allow our product listing to work just like it did before and our order page to have the smarter sufficient-quantity check.

Test it out and, when it works, we're done with iteration 5!

Iteration 6: Establishing Identity

What's the point of a web application if only one person can use it? Let's make our system support multiple users. There are three pieces to making this happen:

- **Authentication** – Establish identity
- **Ownership** – Attach data records to user records
- **Authorization** – Control who is allowed to do what

Background on Authentication

There have been about a dozen popular methods for authenticating Rails applications over the past five years.

The most popular right now is Devise [https://github.com/plataformatec/devise] because it makes it very easy to get up and running quickly. The downside is that the implementation uses very aggressive Ruby metaprogramming techniques which make it very challenging to customize.

In the past I've been a fan of AuthLogic [https://github.com/binarylogic/authlogic] because it takes a very straightforward model/view/controller approach, but it means you have to write a lot of code to get it up and running.

As we learn more about constructing web applications there is a greater emphasis on decoupling components. It makes a lot of sense to depend on an external service for our authentication, then that service can serve this application along with many others.

Introducing OmniAuth

The best application of this concept is the OmniAuth [https://github.com/intridea/omniauth] . It's popular because it allows you to use multiple third-party services to authenticate, but it is really a pattern for component-based authentication.

You could let your users login with their Twitter account, but you could also build your own OmniAuth provider that authenticates all your company's apps. Maybe you can use the existing LDAP provider to hook into ActiveDirectory or OpenLDAP, or make use of the Google Apps interface?

Better yet, OmniAuth can handle multiple concurrent strategies, so you can offer users multiple ways to authenticate. Your app is just built against the OmniAuth interface, those external components can be changed later.

Getting Started with OmniAuth

The first step is to add the dependency to your `Gemfile`:

```
gem "omniauth-twitter"
```

Then run `bundle` from your terminal.

OmniAuth runs as a "Rack Middleware" which means it's not really a part of our app, it's a thin layer between our app and the client. To instantiate and control the middleware, we need an initializer. Create a file `/config/initializers/omniauth.rb` and add the following:

```
Rails.application.config.middleware.use OmniAuth::Builder do
  provider :twitter, "i0KU4jLYYjWzxpTraWviw", "djMu7QStnBK89MkfIg78t
end
```

What is all that garbage? Twitter, like many API-providing services, wants to track who's using it. They accomplish this by distributing API accounts. Specifically, they use the OAuth protocol which requires a "comsumer key" and a "consumer secret." If you want to build an application using the Twitter API you'll need to register and get your own credentials [https://dev.twitter.com/apps] . For this tutorial, I've registered a sample application and given you my key/secret above.

Trying It Out

You need to **restart your server** so the new library and initializer are picked up. In your browser go to `http://127.0.0.1:3000/auth/twitter` and, after a few seconds, you should see a Twitter login page. Login to Twitter using any account, then you should see a **Routing Error** from your application. If you've got that, then things are on the right track.

If you get to this point and encounter a **401 Unauthorized** message there is more work to do. You're probably using your own API key and secret. You need to go into the settings on Twitter for your application [https://dev.twitter.com/apps/] , and add `http://127.0.0.1` as a registered callback domain. I also add `http://0.0.0.0` and `http://localhost` while I'm in there. Now give it a try and you should get the **Routing Error**

Handling the Callback

The way this authentication works is that your app redirects to the third party authenticator, the third party processes the authentication, then it sends the user back to your application at a "callback URL". Twitter is attempting to send the data back to your application, but your app isn't listening at the default OmniAuth callback address, `/auth/twitter/callback`. Let's add a route to listen for those requests.

Open `config/routes.rb` and add this line:

```
match '/auth/:provider/callback', to: 'sessions#create', via: :get
```

Re-visit `http://localhost:3000/auth/twitter`, it will process your already-existing Twitter login, then redirect back to your application and give you **Uninitialized Constant SessionsController**. Our router is attempting to call the `create` action of the `SessionsController`, but that controller doesn't exist yet.

Creating a Sessions Controller

Let's use a generator to create the controller from the command line:

Terminal

```
$ bin/rails generate controller sessions
```

Then open up that controller and add code so it looks like this:

```
class SessionsController < ApplicationController
  def create
    render text: request.env["omniauth.auth"]
  end
end
```

Revisit `/auth/twitter` and, once it redirects to your application, you should see a bunch of information provided by Twitter about the authenticated user. Now we just need to figure out what to **do** with all that.

Creating a User Model

Even though we're using an external service for authentication, we'll still need to keep track of user objects within our system. Let's create a model that will be responsible for that data.

As you saw, Twitter gives us a ton of data about the user. What should we store in our database? The minimum expectations for an OmniAuth provider are three things:

- **provider** – A string name uniquely identifying the provider service
- **uid** – An identifying string uniquely identifying the user within that provider
- **name** – Some kind of human-meaningful name for the user

Let's start with just those three in our model. From your terminal:

Terminal

```
$ bin/rails generate model User provider:string uid:string na
```

Then update the database with `bin/rake db:migrate`.

Creating Actual Users

How you create users might vary depending on the application. For the purposes of our shopping cart, we'll allow anyone to create an account automatically just by logging in with the third party service.

Finding or Creating User Objects

Hop back to the `SessionsController`. I believe strongly that the controller should have as little code as possible, so we'll proxy the User lookup/creation from the controller down to the model like this:

```ruby
def create
  @user = User.find_or_create_by_auth(request.env["omniauth.auth"])
end
```

Now the `User` model is responsible for figuring out what to do with that big hash of data from Twitter. We encapsulate the complexity at the model layer where it's easy to test an reuse. Open the `User` model and add this method:

```ruby
def self.find_or_create_by_auth(auth_data)
  find_or_create_by_provider_and_uid_and_name(auth_data["provider"],
end
```

Rails supports dynamic finders. The one we've used here is composed of:

- `find_or_create` Attempt to find a match in the database based on the following parameters. If found, return it. Otherwise, initialize a new record with these values and save it.
- `_by_provider_and_uid_and_name` Do the lookup and initialization using the provider (first parameter), the uid (second parameter), and the name (third).

This will work, but it's fragile. The UID and provider aren't going to change, so those are fine. The name might change, though. On Twitter, for instance, the humanreadable name is configurable. If they change their name on the external service this lookup is going to fail, creating a second account for them on our service.

Instead, do the lookup with just the fields that won't change:

```
def self.find_or_create_by_auth(auth_data)
  find_or_create_by_provider_and_uid(auth_data["provider"], auth_dat
end
```

This is fault-tolerant, but it has an issue. When a user visits our site for the first time the user object will be created. But we're now not utilizing the name at all — so new records won't have the human name saved.

Thankfully `find_or_create_by` has a solution. We can add a hash of parameters that will only be used when creating records, not as part of the lookup:

```
def self.find_or_create_by_auth(auth_data)
  find_or_create_by_provider_and_uid(auth_data["provider"], auth_dat
                                     name: auth_data["info"]["name"]
end
```

That will work great!

Create Action Redirection

Now, back to `SessionsController`, let's add a redirect action to send them to the `products_path` after login:

```
def create
  @user = User.find_or_create_by_auth(request.env["omniauth.auth"])
  session[:user_id] = @user.id
  redirect_to products_path, notice: "Logged in as #{@user.name}"
end
```

Now visit `/auth/twitter` and you should eventually be redirected to your Products listing and the flash message at the top will show a message saying that you're logged in.

UI for Login/Logout

That's exciting, but now we need links for login/logout that don't require manually manipulating URLs. Anything like login/logout that you want visible on every page goes in the layout.

Open `app/views/layouts/application.html.erb` and you'll see the framing for all our view templates. Let's add in the following **just below the flash message**:

```
<div id="account">
  <% if current_user %>
    <span>Welcome, <%= current_user.name %></span>
    <%= link_to "logout", logout_path, id: "login" %>
  <% else %>
    <%= link_to "login", login_path, id: "logout" %>
  <% end %>
</div>
```

If you refresh your browser that will all crash for several reasons.

Accessing the Current User

It’s a convention that Rails authentication systems provide a `current_user` method to access the user. Let’s create that in our `ApplicationController` with these steps:

- Underneath the `protect_from_forgery` line, add this: `helper_method :current_user`
- Just before the closing `end` of the class, add this:

```
private
  def current_user
    @current_user ||= User.find(session[:user_id]) if session[:user_
  end
```

Even though the `current_user` method is private in `ApplicationController`, it will be available to all our controllers because they inherit from `ApplicationController`. In addition, the `helper_method` line makes the method available to all our views. Now we can access `current_user` from any controller and any view!

Refresh your page and you’ll move on to the next error, `undefined local variable or method `login_path'`.

Convenience Routes

Just because we’re following the REST convention doesn’t mean we can’t also create our own named routes. The view snippet we wrote is attempting to link to `login_path` and `logout_path`, but our application doesn’t yet know about those routes.

Open `config/routes.rb` and add two custom routes:

```
match "/login" => redirect("/auth/twitter"), as: :login, via: :get
match "/logout" => "sessions#destroy", as: :logout, via: :get
```

The first line creates a path named `login` which just redirects to the static address `/auth/twitter` which will be intercepted by the OmniAuth middleware. The second line creates a `logout` path which will call the destroy action of our `SessionsController`.

With those in place, refresh your browser and it should load without error.

We still don’t quite get logged in when we click login, though.

We need to set the session variable in the `SessionsController#create` action:

```ruby
def create
  @user = User.find_or_create_by_auth(request.env["omniauth.auth"])
  session[:user_id] = @user.id
  redirect_to products_path, notice: "Logged in as #{@user.name}"
end
```

Implementing Logout

Our login works great, but we can't logout. When you click the logout link it's attempting to call the `destroy` action of `SessionsController`. Let's implement that.

- Open `SessionsController`
- Add a `destroy` method
- In the method, erase the session marker by setting `session[:user_id] = nil`
- Redirect them to the `root_path` with the notice `"Goodbye!"`
- Define a `root_path` in your router like this: `root to: "products#index"`

Now try logging out and you'll be taken to the products index page.

Connecting Users to Orders

We've done the hard work of creating `User` objects and logging them into the system. Now we need to tie orders to users.

Connecting the Order to a User

Open the `Order` model and add a new `belongs_to` line:

```ruby
belongs_to :user
```

Connecting the User to Orders

Then in the `User` model, just add `has_many :orders`

User/Order Workflow

Now we need to figure out when in the lifecycle we can connect a `User` and an `Order`

1. When an order is created, connect them to the current user if one is logged in
2. When they login, connect them to the current order

Displaying the User on the Order

Let's output the associated `User` on the order's `show` page. There's already a TH for "Customer", just add this into the TD:

```erb
<td><%= @order.user.name if @order.user %></td>
```

When an Order is Created

Our orders are created in the `load_order` method in `OrderItemsController`. Let's modify it to associate a new order with the current user, if one is logged in:

```
def load_order
  @order = Order.find_or_initialize_by_id(session[:order_id],
                                          status: "unsubmitted", u
  if @order.new_record?
    @order.save!
    session[:order_id] = @order.id
  end
end
```

When an Order Exists before They Login

Open up the `SessionsController`. The `create` action is what logs them in. Once the `User` is known we can look for an existing order and, if there is one, connect it to the `User`:

```
def create
  @user = User.find_or_create_by_auth(request.env["omniauth.auth"])
  session[:user_id] = @user.id
  load_order
  @order.update_attributes(user: @user)
  redirect_to products_path, notice: "Logged in as #{@user.name}"
end
```

Test it by logging out, adding a product to your cart, and then logging in.

It blows up, because it doesn't know about `load_order`.

If we move `load_order` into the `ApplicationController` it will be accessible to both the OrderItemsController and the SessionsController.

Clear the Order on Logout

As long as we're in the `SessionsController`, let's clear the order from the session when they logout like this:

```
def destroy
  session[:user_id] = nil
  session[:order_id] = nil
  redirect_to root_path
end
```

Test It!

Now you should be all set. Try creating orders when you're not logged in, then login and the order is preserved. Login first, then create an order and it's connected to your account.

Iteration 7: Checkout

We've got a decent shopping experience going — except you can't actually place the order.

Let's build the ability to add a shipping address and "submit" the order. We won't actually integrate with a payment gateway, but can point out along the way where that would happen.

Building Addresses

A user might ship different orders to different places, so the address needs to be associated with the order. But, at the same time, our frequent customers don't want to enter in the same address every time.

Designing the Data Relationships

The solution? We'll build an address model that is tied to a user. When the user submits the order, they'll pick one of their addresses or add a new one. The order will then be connected to that address.

From a model/database perspective:

- A `User` will have many `Addresses`
- An `Address` will belong to a `User`
- An `Order` will belong to an `Address`
- An `Address` will have many `Orders`

The `belongs_to` side of a relationship holds the foreign key. So we see that `Address` needs a foreign key `user_id` and the `Order` needs an `address_id`.

The Address Model

We know that it'll need a `user_id`, but what else goes into a US address?

- Street Number & Street Name ("Line 1")
- Apartment / Suite ("Line 2")
- City
- State
- Zipcode

All those fields can be stored as strings. Let's use the scaffold generator, even though we won't use all the parts:

Terminal

```
$ bin/rails generate scaffold Address line1:string line2:stri
  bin/rake db:migrate
$
```

Validating Addresses

Everything except `line2` should be required in an address. Let's assume that the zipcode should be exactly 5 characters that are only digits. The `state` must be a two-letter uppercase abbreviation.

Add validations that protect each of these requirements. Look here for tips: Rails Guide on
Validations [http://guides.rubyonrails.org/active_record_validations.html]

Modifying the Orders Table

We also need to add that `address_id` foreign key to the orders table. That means creating a migration.

When you generate a migration to add a column to an existing database you can do everything from the command line if you follow the convention. It goes like this:

Terminal

```
$ bin/rails generate migration add_[name]_to_[table] [column_
```

So in this case:

Terminal

```
$ bin/rails generate migration add_address_id_to_orders addre
$ bin/rake db:migrate
```

Our database is setup, but there is a lot more to do!

Setting Relationships

Open the `Order`, `User`, and `Address` models. Add the relationships we described above.

Checkout UI

Log in, go to the products page, and add something to your cart.

When a user is on the order page they should be able to select an existing address from a drop-down list of their associated addresses or click a link to add a new address.

Building Addresses in the Console

Let's create two sample addresses for your current user in the console. Here's how you might do that:

IRB

```
2.1.1 :001> a = User.last.addresses.build(line1: "123 First St
            a.save
```

```
2.1.1 :002>  b = User.last.addresses.build(line1: "321 Second St
2.1.1 :003>  b.save

2.1.1 :004>
```

Wrapping the Order in a Form

Open `app/views/orders/show.html.erb` and add another row to the end of the table like this:

```
<tr>
  <th>Shipping To:</th>
  <td>
    (addresses select box)
    (add an address link)
  </td>
</tr>
```

To have a select box and, eventually, a "submit" button we'll need a form. The best approach is probably to enclose the entire table in the form tags. Up above the table opening tag, add a `form_for` like this:

```
<%= form_for @order do |f| %>
```

Then put a matching `end` at the bottom of the template.

Displaying Addresses on the Order

Now to figure out that select. Creating select boxes in Rails has always seemed unnecessarily difficult.

We'll use the form helper `select`. It expects as parameters the name of the attribute being set, here `address_id` then an array of options. Each of those options should be a two element array where the first element is the name to show on the form and the second is the value to submit. In this case, we want:

```
<%= f.select :address_id, current_user.addresses.collect{|a| [a.to_s
```

The options here are built by grabbing all the current user's addresses and creating an array made up of arrays containing the address converted to a string (`to_s`) and the `id`.

Load it in your browser and you should see something like `#<Address:0x0000028394482` in the select box. This is actually good!

Implementing the `to_s`

There are a few ways we could format the address for presentation in the select box, but the easiest is to define `to_s` in the model. Whenever you see that format like

`#<ClassName:0x00002838203` you're looking at a model that doesn't implement `to_s`, but `to_s` is being called on it anyway. This implementation of `to_s` comes from Ruby's `Object`, the parent of all objects.

Let's implement a better `to_s` in the `Address` model. Define the method, build an array of the attributes you want to display, then join them together with a comma and a space.

If you have sample addresses without a "Line 2", you'll see an extra comma in the output. You can remove `nil` or empty string entries with this method call: `compact` on the array of attributes.

Adding Addresses

Now let's build out that "add an address" link. You can handle this on your own, here are the steps:

- Write a `link_to` that points to the `new_address_path`
- In the `new` action of `AddressesController`, use `current_user` to build the new object
- On the form, make the `user_id` field hidden
- If the `save` succeeds, redirect back to the order.

Submitting the Order

We can pick an address, but we can't submit the order. Add another row to the table that includes a submit button like this:

```
<%= f.submit "Submit Order" %>
```

View it in your browser, click the button, and look at the log file from your server.

You'll see that it got a PATCH request to `"/orders/1"`, and that it gets processed by the OrdersController#update action.

The action succeeds, but there's a warning in the log file:

```
Unpermitted parameters: address_id
```

We'd like to include `address_id` in the list of permitted order_params at the bottom of the `orders_controller.rb` file:

```
def order_params
  params.require(:order).permit(:user_id, :status, :address_id)
end
```

Changing the Order Status

```
def update
```

```
  respond_to do |format|
    if @order.update(order_params.merge(status: 'submitted'))
      # ...
    else
      # ...
    end
  end
end
```

Cleaning Up

After updating the order, remove order_id from the session so they can't edit it:

```
session[:order_id] = nil
```

Redirect to Confirmation Page

Lastly, we want to redirect to a thank you page with an order summary.

```
format.html { redirect_to confirm_order_path(@order) }
```

If you submit an order, you'll get an `undefined local variable or method `confirm_order_path'` error.

Open up `config/routes.rb` and update the resources for orders:

```
resources :orders do
  member do
    get :confirm
  end
end
```

Create an empty action in `OrdersController` called `confirm`. It is going to need to get the order, so add `:confirm` to the `before_action` for `set_order` at the top of the file.

Create a template in `app/views/orders/confirm.html.erb`.

Go ahead and flesh that page out.

It Should Work!

You should now be able to submit and order. Iteration 7 is complete!

Next Steps

Here are some extension ideas:

- Add authorization so only administrators can add or change product information
- Create an admin interface for viewing and modifying all orders

- Create a `/profile` page where a user can view/change their information and past orders
- Let users pick a default shipping address
- Add a billing address that uses the same set of user addresses