Unit 6: E-Commerce App

[Lesson 6.1: Products, Categories, and Brands](#h.c8uto9a0rby8)

[Gem of the Day](#h.z1ew4uzcr1jf)

[Our Checklist](#h.ud2i55ubgsha)

[Feature Branch](#h.yu3ezkp6q4me)

[Resources and Model](#h.oergoyidmvxo)

[Scaffolds](#h.nlghpph7bc1o)

[Backroom vs Storefront Views](#h.t18cq7hgshjp)

[Backroom](#h.y1ldwgv7fwdb)

[Add Paperclip Images to Products](#h.z1mq7sniamh1)

[Updating Product View](#h.sizl1x9p380o)

[Products: Data Entry](#h.cqw8zrqk7osk)

[Storefront](#h.adzbike5skf3)

[All Items View](#h.fwas5njb27vk)

[Bootstrap Modals](#h.ojijbfkn9u7t)

[Homework](#h.kdh81q3n01fs)

[Lesson 6.2: Carts (Sessions) and Line Items](#h.r1rl1u9amba8)

[Gem of the Day](#h.4ae365jxvgpm)

[Navbar and Shop Listing by Category](#h.6bxalac9rbpf)

[Shopping Cart Set Up](#h.56f8vpnqhn9x)

[Cart Model](#h.vyle25sbe21k)

[Line Items](#h.6916v0spyzfp)

[Add to Cart Button](#h.71svnjjcfye0)

[Cart View](#h.a5hn0xu6hdqc)

[Multiple Items of Same Product](#h.1pmee3hgqd66)

[Emptying Cart](#h.idinvgxvokne)

[Homework](#h.qqr1pa89uyis)

[Lesson 6.3: Cart Security & Orders](#h.imim044j4wsb)

[Gem of the Day](#h.hl1sq9jaj0ln)

[Review: What We Did So Far](#h.6mw2vff0i5wd)

[Security](#h.gfiy9g54qkxv)

[Checking Out](#h.b30jxdfy5qvj)

[What is an Order?](#h.i9ohjjv6et67)

[Checkout Button](#h.p0uogj4z24dh)

[New Order (Checkout) View](#h.urqhsprguo9o)

[Creating the Order](#h.kso5y58lnbn7)

[Homework](#h.i3mpj7rf4yo4)

# Lesson 6.1: Products, Categories, and Brands

Let’s add e-commerce capability to our Doggy Daycare app.

## Gem of the Day

Shut the F\*ck Up <https://github.com/tpope/gem-shut-the-fuck-up>

Roll Call

Review last class - what gems? what learned about devise? what about alternatives?

## Our Checklist

* Let's start with a customer-free store (as shady as that sounds).
* We'll have products that are shown all together or by category.
* You can buy one or more of each product, and we'll add them to a cart.
* Then we'll tie a user to that cart ("check out").
* At the end we'll learn how to implement a payment process.

## Feature Branch

First, create a feature branch called “shop” that we will use as we develop this feature over the next few days:

git checkout -b shop

## Resources and Model

Product

* Name
* Price
* Quantity
* Description
* Brand?
* category\_id?

Category

* Name

We won't have a Cart resource, exactly, but two separate resources: one to store an item in as you shop, the other gather those items together:

LineItem

* product\_id
* quantity
* line\_item\_total
* order\_id

Order

* sub\_total
* sales\_tax
* grand\_total
* customer\_id\*

\* how we'll eventually tie a user (customer) to what they want to buy

### Scaffolds

* rails g scaffold Category name:string
* rails g scaffold Product name:string price:decimal quantity:integer description:text brand:string rating:integer category:references
* rake db:migrate
* associate products and categories (has many, belongs to)

## Backroom vs Storefront Views

We will use the scaffolded Views as our "backroom", a place for employees (admin users) can create and update products and categories.

We'll create new views (within a new controller) to act as our "storefront" - views for our customers to peruse products (all or by category) and purchase.

### Backroom

Add Bootstrap table class to the scaffold index table for products and categories.

Then, seed categories:

Breed.create(breed: 'Australian Shepherd')

Breed.create(breed: 'Border Collie')

Breed.create(breed: 'Jack Russell')

Breed.create(breed: 'Pit Bull')

Breed.create(breed: 'Shih-tzu')

Breed.create(breed: 'Standard Poodle')

Breed.create(breed: 'Miniature Poodle')

Breed.create(breed: 'Doberman')

Breed.create(breed: 'Bulldog')

Breed.create(breed: 'Alaskan Malamute')

Breed.create(breed: 'Eskimo')

Breed.create(breed: 'Chihuahua')

Breed.create(breed: 'Yorkie')

Breed.create(breed: 'Boxer')

Breed.create(breed: 'Mutt')

Owner.create(first\_name: "Sia", last\_name: "Karamalegos", phone: "504-555-1212", emerg\_name: "Gail Karamalegos", emerg\_phone: "979-555-1212")

Dog.create(name: "Priscilla", owner\_id: 1, breed\_id: 1, vet: "Magazine Street Animal Clinic", in\_daycare: true)

Dog.create(name: "Harry", owner\_id: 1, breed\_id: 3, vet: "Magazine Street Animal Clinic", in\_daycare: false)

categories = Category.create([{name: 'toys'}, {name: 'treats'}, {name: 'beds'}, {name: 'food'}])

Run rake db:drop, then rake db:setup (migrates and seeds).

## Add Paperclip Images to Products

rails g paperclip product image

rake db:migrate

Add validations to products model:

has\_attached\_file :image, :styles => { :medium => "300x300#", :thumb => "100x100#" }, :default\_url => "missing\_product\_:style.png"

validates\_attachment\_content\_type :image, :content\_type => /\Aimage\/.\*\Z/

Add :image to the product\_params at the bottom of the Products controller.

## Updating Product View

In the Product form, we'll remove the div for taking a rating (the customer's will be given that power later), and we'll put in a field for taking an uploaded image, and we'll tweak the field that takes the category\_id so that it's a drop-down menu to select a category.

<div class="field form-group">

<%= f.label :image %><br>

<%= f.file\_field :image %>

</div>

<div class="field form-group">

<%= f.label :category\_id %><br>

<%= f.collection\_select :category\_id, Category.all, :id, :name, {prompt: "Select a category"}, {class: "form-control"} %>

</div>

### Products: Data Entry

Now you can go through and make a couple products for us to sell!

One hint for improvement: put "number\_to\_currency" (in the Ruby tags) in front of the product.price to add a dollar sign and round to two decimal places, like so:

<%= number\_to\_currency product.price %>

## Storefront

Let's give the customers three ways to view products:

* All at once (like the Products index page)
* By Category (will use parameters to filter all items)
* By Brand (the Product attribute - will set up later maybe using JavaScript or AJAX)

$ rails g controller Storefront all\_items

### All Items View

Since customers will see this, let’s make it a bit prettier than our back-office index pages:

* The "+ quick info" link will bring up a modal with product highlights.
* Star/Rating system is commented out until we set that for the Users.

First, add to the Storefront controller, all\_items method:

@products = Product.all

Then in the all\_items view:

<h1>Welcome to Fur-Baby Fun-Time Store!</h1>

<p>Where you can spend all your money on your fur babies!</p>

<div class="row">

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

<div class="col-md-4">

<div class="panel panel-default height">

<div class="panel-body">

<%= image\_tag product.image.url(:thumb), class: "img-responsive" %>

<p><a href="#">

<span class="glyphicon glyphicon-plus"></span> quick info

</a></p>

<p><strong><%= number\_to\_currency product.price %></strong></p>

<p><%= product.name %></p>

<p>Made by: <%= product.brand %></p>

<p>

<%# product.rating.times do %>

<!-- <span class="glyphicon glyphicon-star yellow"></span> -->

<%# end %>

</p>

</div>

</div>

</div>

<% end %>

</div>

Then in the storefront.scss:

.height {

height: 260px;

}

### Bootstrap Modals

Now let’s get those quick info modals working!

Let's test a demo modal on our page before fitting it into our product loop.

Go to: <http://getbootstrap.com/javascript/#modals>

Copy the code from the "Live Demo" section, and paste it in the bottom of your all\_items.html.erb file:

<!-- Button trigger modal -->

<button type="button" class="btn btn-primary btn-lg" data-toggle="modal" data-target="#myModal">

Launch demo modal

</button>

<!-- Modal -->

<div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal" aria-label="Close"><span aria-hidden="true">&times;</span></button>

<h4 class="modal-title" id="myModalLabel">Modal title</h4>

</div>

<div class="modal-body">

...

</div>

<div class="modal-footer">

<button type="button" class="btn btn-default" data-dismiss="modal">Close</button>

<button type="button" class="btn btn-primary">Save changes</button>

</div>

</div>

</div>

</div>

Check that it works (just mostly blank modal pops up)

Let's use that modal code to create our own "quick info" modal. First, take the functionality from the demo button and move it to the anchor tags around "+ quick info". In addition, add "\_<%= product.id %>" to the data-target, this will let us have a unique element ID each time we loop through.

<!-- plus button trigger modal -->

<p><a href="#" data-toggle="modal" data-target="#myModal\_<%= product.id %>">

<span class="glyphicon glyphicon-plus"></span> quick info

</a></p>

Next move the modal code into a space between the <% end %> tag and the last </div> to close above that. At the top of this code you'll find id="myModal" add "\_<%= product.id %>" to that so it corresponds to the data-target in the anchor tag above. In addition, you will replace the modal title with the product name, and add the product price and description in the body:

</div> <!-- col-md-4 -->

<!-- Modal -->

<div class="modal fade" id="myModal\_<%= product.id %>" tabindex="-1" role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal" aria-label="Close"><span aria-hidden="true">&times;</span></button>

<h4 class="modal-title" id="myModalLabel"><%= product.name %></h4>

</div>

<div class="modal-body">

<p><%= product.price %></p>

<p><%= product.description %></p>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-default" data-dismiss="modal">Close</button>

<button type="button" class="btn btn-primary">Save changes</button>

</div>

</div>

</div>

</div> <!-- modal -->

<% end %>

Check that your modals are working, and then commit your changes!

## Homework

Homework: Add the medium photo to the modal. Make your views prettier. Move sign-in/sign-out links to right on navbar. Optional: Move dog search box to dogs index page.

# Lesson 6.2: Carts (Sessions) and Line Items

## Gem of the Day

Friendly ID <https://github.com/norman/friendly_id>

## Navbar and Shop Listing by Category

Let’s update the navbar to include links to our storefront. The first thing we will do is update the logic in our controller to handle a cat\_id parameter that will modify what is in @products.

def all\_items

if params[:cat\_id]

@category = Category.find(params[:cat\_id])

@products = Product.where(category\_id: params[:cat\_id])

else

@products = Product.all

end

end

Now, let’s add our shop dropdown menu right after our home page link in the navbar - we are creating links to the all\_items page and passing in a parameter for the category ID which we will then use to only show those items on that page.

<li class="dropdown">

<a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-expanded="false">Shop<span class="caret"></span></a>

<ul class="dropdown-menu" role="menu">

<% Category.all.each do |cat| %>

<li><%= link\_to cat.name.capitalize, storefront\_all\_items\_path(cat\_id: cat.id) %></li>

<% end %>

<li class="divider"></li>

<li><%= link\_to "All Items", storefront\_all\_items\_path %></li>

</ul>

</li>

Finally, let’s add a heading2 to list which category is being shown on the all\_items page:

<% if params[:cat\_id] %>

<h3>All <%= @category.name.capitalize %></h3>

<% else %>

<h3>All Items</h3>

<% end %>

Commit your changes!

## Shopping Cart Set Up

Yesterday, we mentioned that our cart would be handled by 2 resources - orders and line items:

LineItem

* product\_id
* quantity
* line\_item\_total
* order\_id

Order

* sub\_total
* sales\_tax
* grand\_total
* customer\_id\*

\* how we'll eventually tie a user (customer) to what they want to buy

### Cart Model

Well, it’s a bit more complicated than that. We also need a temporary cart that will store info from the current session, then when they checkout, the app will generate the line items and order. To do this, we’ll keep a cart in the database and store its unique identifier, card.id, in the **session**. Every time a request comes in, we can recover the identity from the session and use it to find the cart in the database. We’ll talk more about sessions once we implement one little bit more.

To create the cart:

rails g scaffold Cart

rake db:migrate

Rails makes the current session look like a hash to the controller, so we will store the ID of the cart in the session by indexing it with the symbol :cart\_id. To do this, we create a new file in app/controllers/concerns called **current\_cart.rb**:

module CurrentCart

extend ActiveSupport::Concern

private

def set\_cart

@cart = Cart.find(session[:cart\_id])

rescue ActiveRecord::RecordNotFound

@cart = Cart.create

session[:cart\_id] = @cart.id

end

end

The set\_cart method essentially grabs the cart\_id from the session object and then attempts to find a cart associated with that id. If one doesn’t exist, it creates a new cart.

We placed this method in a CurrentCart module and marked it as private so that we can share this common code among controller and prevent Rails from making it an action on the controller.

### Line Items

A cart will contain the line items until we generate the order. Let’s scaffold the line items now:

rails g scaffold LineItem product:references cart:belongs\_to quantity:integer

Before migrating, edit the migration so that the quantity defaults to 1:

t.integer :quantity, default: 1

Now, run rake db:migrate.

At the model level, there is no difference between a simple reference and a “belongs\_to” relationship.

Let’s finish setting up the other side of the relationship in the Cart model:

class Cart < ActiveRecord::Base

has\_many :line\_items, dependent: :destroy

end

The dependent: :destroy part indicates that the existence of the line items is dependent on the existence of the cart. If we destroy a cart (delete it from the database), then we want Rails to also destroy any line items associated with that cart.

Finally, let’s set up the other side of the relationship in the Product model:

class Product < ActiveRecord::Base

belongs\_to :category

has\_many :line\_items

before\_destroy :ensure\_not\_referenced\_by\_any\_line\_item

...

private

# ensure that there are no line items referencing this product

def ensure\_not\_referenced\_by\_any\_line\_item

if line\_items.empty?

return true

else

errors.add(:base, 'Line Items present')

return false

end

end

end

end

Here we defined a **hook** method that Rails will call automatically whenever Rails attempts to destroy a row for Product in the database. The **errors** object is the same place that data validation errors are stored.

### Add to Cart Button

Now let’s add an Add to Cart button for each product. Because we scaffolded Line Items, we can use the Create method that the scaffold generated. Each time we add to a cart, we are actually creating Line Items. Rails will handle creating the cart if it does not already exist.

When we talked about HTTP verbs and the scaffold methods, we learned that the create method uses the POST verb. The Rails link\_to method uses GET by default, but the button\_to method uses POST, so we will use that here.

So, in the all\_items view, just add this button to the panel:

<%= button\_to 'Add to Cart', line\_items\_path(product\_id: product), class: "btn btn-success btn-sm" %>

We have to make a few changes before that functionality will work. In the **Line Items Controller**, we need to find the shopping cart for the current session or create one if it doesn’t already exist. Then, we need to add that product to the cart and display the cart contents. We built part of this code in our CurrentCart module, but now we need to use it.

class LineItemsController < ApplicationController

include CurrentCart

before\_action :set\_cart, only: [:create]

Now, scroll down to the create method so that we can add just a few lines of code to recognize the product\_id parameter and build the cart (we’re also removing the notice about the line item being created):

def create

product = Product.find(params[:product\_id])

@line\_item = @cart.line\_items.build(product: product)

respond\_to do |format|

if @line\_item.save

format.html { redirect\_to @line\_item.cart }

format.json { render :show, status: :created, location: @line\_item }

else

format.html { render :new }

format.json { render json: @line\_item.errors, status: :unprocessable\_entity }

end

end

end

### Cart View

Last but not least, let’s add some data to the cart show view:

<h2>Your Shopping Cart</h2>

<table class="table">

<thead>

<tr>

<th>Product</th>

<th>Quantity</th>

<th>Unit Price</th>

<th>Total Price</th>

</tr>

</thead>

<tbody>

<% @cart.line\_items.each do |item| %>

<tr>

<td><%= item.product.name %></td>

<td><%= item.quantity %> &times;</td>

<td><%= number\_to\_currency item.product.price %></td>

<td><%= number\_to\_currency item.total\_price %></td>

</tr>

<% end %>

</tbody>

</table>

<h3>Order Subtotal: <%= number\_to\_currency @cart.subtotal %></h3>

<%= link\_to "Continue Shopping", storefront\_all\_items\_path, class: "btn btn-success btn-sm" %>

<%= button\_to "Empty Cart", @cart, method: :delete, data: {confirm: "Are you sure?"}, class: "btn btn-danger btn-sm", form: {style: 'display:inline-block;'} %>

To make this work, we need to create the **total\_price** methods for both line items and carts, in their respective models:

class LineItem < ActiveRecord::Base

...

def total\_price

product.price \* quantity

end

end

class Cart < ActiveRecord::Base

...

def subtotal

line\_items.to\_a.sum { |item| item.total\_price }

end

end

Go ahead and run your server and test out adding an item to the cart. If everything works as expected, commit your changes!

### Multiple Items of Same Product

Now, let’s make it so that when we add the same product to the cart, it doesn’t generate a new line item, it only updates the quantity. We need an **add\_product** method in our **Cart Model** to handle this:

class Cart < ActiveRecord::Base

has\_many :line\_items, dependent: :destroy

def add\_product(product\_id)

current\_item = line\_items.find\_by(product\_id: product\_id)

if current\_item

current\_item.quantity += 1

else

current\_item = line\_items.build(product\_id: product\_id)

end

current\_item

end

end

We also need to update the **create** method in the **Line Items Controller** to take in this method:

def create

product = Product.find(params[:product\_id])

@line\_item = @cart.add\_product(product.id)

Now when we add a duplicate product, it updates the quantity on the line item rather than creating a new line.

### Emptying Cart

Let’s update the redirect after emptying a cart. Go to the Carts Controller, destroy method. We are going to make sure that the user is destroying their own cart first, and then remove the cart from the session before redirecting to all\_items

def destroy

@cart.destroy if @cart.id == session[:cart\_id]

session[:cart\_id] = nil

respond\_to do |format|

format.html { redirect\_to storefront\_all\_items\_url, notice: 'Your cart is currently empty.' }

format.json { head :no\_content }

end

end

Commit your changes!

## Homework

Make things prettier. Add the ability to delete individual line items from the cart - add buttons to each row that are linked to the destroy method for Line Items. Optional: Change dog show views to modals on Dog index.

Learn more about sessions here: <http://www.justinweiss.com/blog/2015/03/17/how-rails-sessions-work/>

Learn more about session, cookies, and authentication here: <http://www.theodinproject.com/ruby-on-rails/sessions-cookies-and-authentication>

# Lesson 6.3: Cart Security & Orders

## Gem of the Day

Stripe vs Active Merchant: <https://www.ruby-toolbox.com/categories/Payments>

Gant is a big fan of Stripe - it’s popular and a gateway.

Stripe: <https://github.com/stripe/stripe-ruby>

Active Merchant: <https://github.com/Shopify/active_merchant>

## Review: What We Did So Far

6.1

* Branch shop
* Products and paperclip image
* Categories and seeds
* All items view with panels and modals

6.2

* Navbar and filter categories
* Cart scaffold
* Module for setting cart
* Line items scaffold, relationships, validations to prevent orphan records
* Add cart button (line item create which also uses the set cart method from the module)
* Cart view and totals
* Fixed multiples

## Security

Let’s take a minute to talk about security and do a few things to make our app more secure. First, try navigating to a cart that doesn’t exist:

<http://localhost:3000/carts/fakecart>

You can see that the app triggers an error - in production that error wouldn’t be clear to the user. So, let’s write some code to trigger some messages in the Carts Controller:

class CartsController < ApplicationController

before\_action :set\_cart, only: [:show, :edit, :update, :destroy]

rescue\_from ActiveRecord::RecordNotFound, with: :invalid\_cart

…

private

…

def invalid\_cart

logger.error "Attempt to access invalid cart #{params[:id]}"

redirect\_to storefront\_all\_items\_path, notice: 'Invalid cart'

end

end

Reload that invalid page and confirm that it redirects to the all\_items path, gives an error on the page, and also logs the attempt in your log.

That’s an improvement, but our biggest problem is that currently anyone can access anyone else’s cart as long as they guess a valid id. Let’s change that by **disallowing the cart id** parameter in the **line items controller**. Scroll down to the line item params and simple delete “, :cart\_id”.

## Checking Out

So, we’ve creating a lot of cool stuff, but a user still can’t check out and actually buy the products in their cart. Today we will add a simplified version of that functionality.

### What is an Order?

An order, in it’s most basic sense, is a set of line items along with details of the purchase transaction like customer contact information and payment option.

Let’s create the Order scaffold and add order ID to the Line Items model:

rails g scaffold Order name:string address:text pay\_type:string user:references

rails g migration add\_order\_to\_line\_item order:references

Double-check your migrations, then run rake db:migrate. Then, set up our relationships - remember that dependent destroy will delete all line items of an order if the order itself is deleted:

class **Order** < ActiveRecord::Base

has\_many :line\_items, dependent: :destroy

belongs\_to :user

class **LineItem** < ActiveRecord::Base

belongs\_to :order

belongs\_to :product

belongs\_to :cart

class **User** < ActiveRecord::Base

# Include default devise modules. Others available are:

# :confirmable, :lockable, :timeoutable and :omniauthable

devise :database\_authenticatable, :registerable,

:recoverable, :rememberable, :trackable, :validatable

has\_many :orders

Then, commit your changes!

### Checkout Button

To start the checkout process, we need a user to click a Checkout button from their shopping Cart view. Let’s add this button before the Continue Shopping button on the Cart show view:

<%= link\_to "Checkout", new\_order\_path, class: "btn btn-info btn-sm" %>

Now, we need to jump in the Orders controller to do a few things:

1. make the user log in
2. give the order access to the cart
3. make sure there is something in the cart
4. if there is something in the cart, give them a form to enter checkout details like contact and payment

For #1 and #2, let’s add this to the top of the Orders controller:

class OrdersController < ApplicationController

include CurrentCart

before\_action :set\_cart, only: [:new, :create]

before\_action :authenticate\_user!

For #3 and #4, let’s add some code to the New method in the Order Controller - basically, if nothing is in the cart, we redirect them to the storefront with a notice, and we return immediately - in other words, we exit the checkout:

def new

if @cart.line\_items.empty?

redirect\_to storefront\_all\_items\_url, notice: "Your cart is empty."

return

end

@order = Order.new

@order.user\_id = current\_user.id

end

### New Order (Checkout) View

Now that we can move to the new order form, let’s make it look better. First, let’s go to the Order New view and add an h1 and h3, and edit the back button to go back to the cart rather than the list of orders:

<h1>Checkout</h1>

<h3>Please Enter Your Details</h3>

<%= render 'form' %>

<%= link\_to 'Back to Cart', cart\_path(session[:cart\_id]) %>

Now, let’s create an array of payment types in the Order model:

PAYMENT\_TYPES = ["Check", "Credit Card", "PayPal"]

Then, let’s edit the form itself:

<%= form\_for(@order) do |f| %>

...

<div class="field form-group">

<%= f.label :name %><br>

<%= f.text\_field :name, class: "form-control", placeholder: "Jane Smith" %>

</div>

<div class="field form-group">

<%= f.label :address %><br>

<%= f.text\_area :address, class: "form-control", placeholder: "100 St. Charles Ave\nNew Orleans, LA 70115" %>

</div>

<div class="field form-group">

<%= f.label :pay\_type %><br>

<%= f.select :pay\_type, Order::PAYMENT\_TYPES, {prompt: "Select a payment method"}, {class: "form-control"} %>

</div>

<div class="actions">

<%= f.submit "Place Order", class: "btn btn-success" %>

</div>

<% end %>

Finally, let’s add some validation to make sure all fields are filled out - go to the Order model:

class Order < ActiveRecord::Base

belongs\_to :user

has\_many :line\_items, dependent: :destroy

validates :name, :address, :user\_id, presence: true

PAYMENT\_TYPES = ["Credit Card", "PayPal", "Check", "Monopoly Money"]

validates :pay\_type, inclusion: PAYMENT\_TYPES

end

Commit your changes!

## Creating the Order

Now we need to update the Create method in the Order controller to accommodate our needs:

* Grabbing the form data to populate a new order object
* Add the line items from our cart to the order using a method that we will create in the step after this
* Validate and save the order - if there is a failure, let them edit.
* After order is successfully saved, delete the cart and redisplay the storefront with a notice that the order was placed.

def create

@order = Order.new(order\_params)

@order.user\_id = current\_user.id

@order.add\_line\_items\_from\_cart(@cart)

respond\_to do |format|

if @order.save

Cart.destroy(session[:cart\_id])

session[:cart\_id] = nil

format.html { redirect\_to storefront\_all\_items\_url, notice: 'Thank you for your order!' }

format.json { render :show, status: :created, location: @order }

else

format.html { render :new }

format.json { render json: @order.errors, status: :unprocessable\_entity }

end

end

end

Now, we need to actually create the **add\_line\_items\_from\_cart** method in the Order Model which will do two things while looping through the line\_items:

* railSet the cart\_id to nil to prevent the line\_item from disappearing when we destroy the cart (after we put it in the order).
* Add the line item to the collection of line items for the order

def add\_line\_items\_from\_cart(cart)

cart.line\_items.each do |item|

item.cart\_id = nil

line\_items << item

end

end

Now let’s check that it is working by checking out. First try checking out without entering anything on the new order page. Then, enter info and checkout. That’s nice, but did it do anything? Let’s check in rails console using ap Order.all, then ap LineItem.all. Pretty cool! Let’s commit our changes.

Classroom Challenge:

* Update the orders index to show the user email.
* Update the orders view to list the line items associated with an order.

## Homework

* Add a shopping cart button to navbar. Bonus - only have it show up if it exists.
* Add an Orders link to the navbar. We will change links based on admin status later.
* Think of a personal app you would like to work on over the next 2 weeks.
* If you have not already, finish yesterday’s homework.