In this chapter, you'll see:

- Sessions and session management
- Adding relationships among models
- Adding a button to add a product to a cart

CHAPTER 9

Task D: Cart Creation

Now that we have the ability to display a catalog containing all our wonderful products, it would be nice to be able to sell them. Our customer agrees, so we've jointly decided to implement the shopping cart functionality next. This is going to involve a number of new concepts, including sessions, relationships among models, and adding a button to the view—so let's get started.

Iteration D1: Finding a Cart

As users browse our online catalog, they will (we hope) select products to buy. The convention is that each item selected will be added to a virtual shopping cart, held in our store. At some point, our buyers will have everything they need and will proceed to our site's checkout, where they'll pay for the stuff in their carts.

This means that our application will need to keep track of all the items added to the cart by the buyer. To do that, we'll keep a cart in the database and store its unique identifier, cart.id, in the session. Every time a request comes in, we can recover that identifier from the session and use it to find the cart in the database.

Let's go ahead and create a cart:

```
depot> bin/rails generate scaffold Cart
 ...
depot> bin/rails db:migrate
==  CreateCarts: migrating ====================================================
-- create_table(:carts)
   -> 0.0012s
==  CreateCarts: migrated (0.0014s) ===========================================
```

Rails makes the current session look like a hash to the controller, so we'll store the ID of the cart in the session by indexing it with the :cart_id symbol:

rails7/depot_f/app/controllers/concerns/current_cart.rb
```
module CurrentCart

  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 starts by getting the :cart_id from the session object and then attempts to find a cart corresponding to this ID. If such a cart record isn't found (which will happen if the ID is nil or invalid for any reason), this method will proceed to create a new Cart and then store the ID of the created cart into the session.

Note that we place the set_cart() method in a CurrentCart module and place that module in a new file in the app/controllers/concerns directory.[1] This treatment allows us to share common code (even as little as a single method!) among controllers.

Additionally, we mark the method as private, which prevents Rails from ever making it available as an action on the controller.

Iteration D2: Connecting Products to Carts

We're looking at sessions because we need somewhere to keep our shopping cart. We'll cover sessions in more depth in Rails Sessions, on page 360, but for now let's move on to implement the cart.

Let's keep things simple. A cart contains a set of products. Based on the Initial guess at application data diagram on page 67, combined with a brief chat with our customer, we can now generate the Rails models and populate the migrations to create the corresponding tables:

```
depot> bin/rails generate scaffold LineItem product:references cart:belongs_to
 ...
depot> bin/rails db:migrate
==  CreateLineItems: migrating ================================================
-- create_table(:line_items)
   -> 0.0013s
==  CreateLineItems: migrated (0.0014s) =======================================
```

1. https://signalvnoise.com/posts/3372-put-chubby-models-on-a-diet-with-concerns

The database now has a place to store the references among line items, carts, and products. If you look at the generated definition of the LineItem class, you can see the definitions of these relationships:

rails7/depot_f/app/models/line_item.rb

```
class LineItem < ApplicationRecord
  belongs_to :product
  belongs_to :cart
end
```

The belongs_to() method defines an accessor method—in this case, carts() and products()—but more importantly it tells Rails that rows in line_items are the children of rows in carts and products. No line item can exist unless the corresponding cart and product rows exist. A great rule of thumb for where to put belongs_to declarations is this: if a table has any columns whose values consist of ID values for another table (this concept is known by database designers as *foreign keys*), the corresponding model should have a belongs_to for each.

What do these various declarations do? Basically, they add navigation capabilities to the model objects. Because Rails added the belongs_to declaration to LineItem, we can now retrieve its Product and display the book's title:

```
li = LineItem.find(...)
puts "This line item is for #{li.product.title}"
```

To be able to traverse these relationships in both directions, we need to add some declarations to our model files that specify their inverse relations.

Open the cart.rb file in app/models, and add a call to has_many():

rails7/depot_f/app/models/cart.rb

```
class Cart < ApplicationRecord
  has_many :line_items, dependent: :destroy
end
```

That has_many :line_items part of the directive is fairly self-explanatory: a cart (potentially) has many associated line items. These are linked to the cart because each line item contains a reference to its cart's ID. The dependent: :destroy part indicates that the existence of line items is dependent on the existence of the cart. If we destroy a cart, deleting it from the database, we want Rails also to destroy any line items that are associated with that cart.

Now that the Cart is declared to have many line items, we can reference them (as a collection) from a cart object:

```
cart = Cart.find(...)
puts "This cart has #{cart.line_items.count} line items"
```

Now, for completeness, we should add a has_many directive to our Product model. After all, if we have lots of carts, each product might have many line items referencing it. This time, we make use of validation code to prevent the removal of products that are referenced by line items:

rails7/depot_f/app/models/product.rb

```
class Product < ApplicationRecord
  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
      unless line_items.empty?
        errors.add(:base, 'Line Items present')
        throw :abort
      end
    end
end
```

Here we declare that a product has many line items and define a *hook* method named ensure_not_referenced_by_any_line_item(). A hook method is a method that Rails calls automatically at a given point in an object's life. In this case, the method will be called before Rails attempts to destroy a row in the database. If the hook method throws :abort, the row isn't destroyed.

Note that we have direct access to the errors object. This is the same place that the validates() method stores error messages. Errors can be associated with individual attributes, but in this case we associate the error with the base object.

Before moving on, add a test to ensure that a product in a cart can't be deleted:

rails7/depot_f/test/controllers/products_controller_test.rb

```
test "can't delete product in cart" do
  assert_difference("Product.count", 0) do
    delete product_url(products(:two))
  end

  assert_redirected_to products_url
end

test "should destroy product" do
  assert_difference("Product.count", -1) do
    delete product_url(@product)
  end

  assert_redirected_to products_url
end
```

And change the fixture to make sure that product two is in both carts:

rails7/depot_f/test/fixtures/line_items.yml

```
# Read about fixtures at
# https://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html

one:
  product: two
  cart: one

two:
  product: two
  cart: two
```

We'll have more to say about intermodel relationships starting in Specifying Relationships in Models, on page 310.

Iteration D3: Adding a Button

Now that that's done, it's time to add an Add to Cart button for each product.

We don't need to create a new controller or even a new action. Taking a look at the actions provided by the scaffold generator, we find index(), show(), new(), edit(), create(), update(), and destroy(). The one that matches this operation is create(). (new() may sound similar, but its use is to get a form that's used to solicit input for a subsequent create() action.)

Once this decision is made, the rest follows. What are we creating? Certainly not a Cart or even a Product. What we're creating is a LineItem. Looking at the comment associated with the create() method in app/controllers/line_items_controller.rb, you see that this choice also determines the URL to use (/line_items) and the HTTP method (POST).

This choice even suggests the proper UI control to use. When we added links before, we used link_to(), but links default to using HTTP GET. We want to use POST, so we'll add a button this time; this means we'll be using the button_to() method.

We could connect the button to the line item by specifying the URL, but again we can let Rails take care of this for us by simply appending _path to the controller's name. In this case, we'll use line_items_path.

However, there's a problem with this: how will the line_items_path method know *which* product to add to our cart? We'll need to pass it the ID of the product corresponding to the button. All we need to do is add the :product_id option to the line_items_path() call. We can even pass in the product instance itself—Rails knows to extract the ID from the record in circumstances such as these.

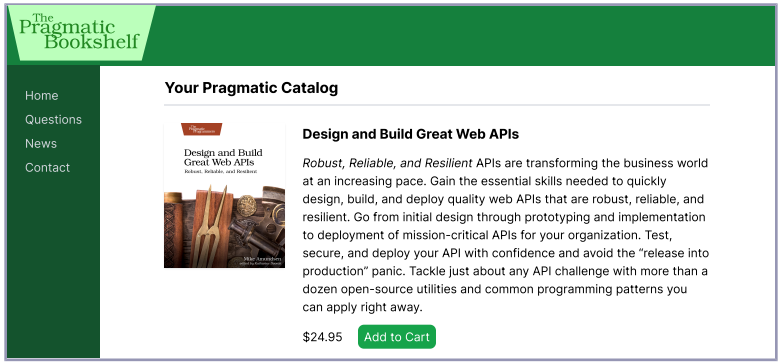
In all, the *one* line that we need to add to our index.html.erb looks like this:

rails7/depot_f/app/views/store/index.html.erb

```
<div class="w-full">
<% if notice.present? %>
  <p class="py-2 px-3 bg-green-50 mb-5 text-green-500 font-medium rounded-lg
            inline-block" id="notice">
    <%= notice %>
  </p>
<% end %>

<h1 class="font-bold text-xl mb-6 pb-2 border-b-2">
  Your Pragmatic Catalog
</h1>

<ul>
  <% cache @products do %>
    <% @products.each do |product| %>
      <% cache product do %>
        <li class='flex mb-6'>
          <%= image_tag(product.image_url,
            class: 'object-contain w-40 h-48 shadow mr-6') %>

          <div>
            <h2 class="font-bold text-lg mb-3"><%= product.title %></h2>

            <p>
              <%= sanitize(product.description) %>
            </p>

            <div class="mt-3">
              <%= number_to_currency(product.price) %>

              <%= button_to 'Add to Cart',
                  line_items_path(product_id: product),
                  form_class: 'inline',
                  class: 'ml-4 rounded-lg py-1 px-2
                          text-white bg-green-600' %>
            </div>
          </div>
        </li>
      <% end %>
    <% end %>
  <% end %>
</ul>
</div>
```

We also need to deal with two formatting issues. button_to creates an HTML <form> wrapping the <button>. HTML <form> is normally a block element that appears on the next line. We'd like to place them next to the price. This is no problem as Rails lets you specify both the form_class as well as the button class.

Now our index page looks like the following screenshot. But before we push the button, we need to modify the create() method in the line items controller to expect a product ID as a form parameter. Here's where we start to see how important the id field is in our models. Rails identifies model objects (and the corresponding database rows) by their id fields. If we pass an ID to create(), we're uniquely identifying the product to add.



Why the create() method? The default HTTP method for a link is a GET, and for a button is a POST. Rails uses these conventions to determine which method to call. Refer to the comments inside the app/controllers/line_items_controller.rb file to see other conventions. We'll be making extensive use of these conventions inside the Depot application.

Now let's modify the LineItemsController to find the shopping cart for the current session (creating one if one isn't there already), add the selected product to that cart, and display the cart contents.

We use the CurrentCart concern we implemented in Iteration D1 on page 116 to find (or create) a cart in the session:

rails7/depot_f/app/controllers/line_items_controller.rb

```
class LineItemsController < ApplicationController
  include CurrentCart
  before_action :set_cart, only: %i[ create ]
  before_action :set_line_item, only: %i[ show edit update destroy ]

  # GET /line_items or /line_items.json
  #...
end
```

We include the CurrentCart module and declare that the set_cart() method is to be involved before the create() action. We explore action callbacks in depth in Callbacks, on page 366, but for now all you need to know is that Rails provides the ability to wire together methods that are to be called before, after, or even around controller actions.

In fact, as you can see, the generated controller already uses this facility to set the value of the @line_item instance variable before the show(), edit(), update(), or destroy() actions are called.

Now that we know that the value of @cart is set to the value of the current cart, all we need to modify is a few lines of code in the create() method in app/controllers/line_items_controller.rb. to build the line item itself:

rails7/depot_f/app/controllers/line_items_controller.rb

```
  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 cart_url(@line_item.cart),
          notice: "Line item was successfully created." }
        format.json { render :show,
          status: :created, location: @line_item }
      else
        format.html { render :new,
          status: :unprocessable_entity }
        format.json { render json: @line_item.errors,
          status: :unprocessable_entity }
      end
    end
  end
```

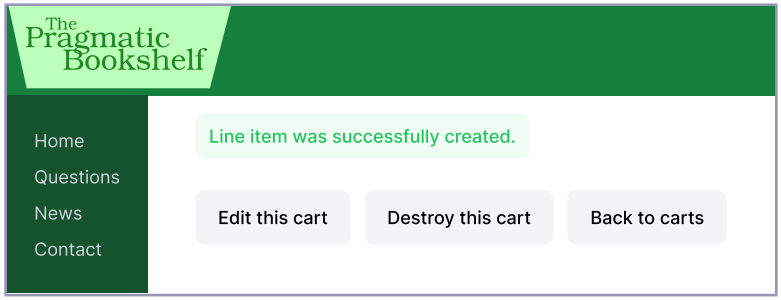
We use the params object to get the :product_id parameter from the request. The params object is important inside Rails applications. It holds all of the parameters passed in a browser request. We store the result in a local variable because there's no need to make this available to the view.

We then pass that product we found into @cart.line_items.build. This causes a new line item relationship to be built between the @cart object and the product. You can build the relationship from either end, and Rails takes care of establishing the connections on both sides.

We save the resulting line item into an instance variable named @line_item.

The remainder of this method takes care of handling errors, which we'll cover in more detail in Iteration E2: Handling Errors, on page 132, (as well as handling JSON requests, which we don't need per se but that were added by the Rails generator). But for now, we want to modify only one more thing: once the line item is created, we want to redirect users to the cart instead of back to the line item. Since the line item object knows how to find the cart object, all we need to do is add .cart to the method call.

Confident that the code works as intended, we try the Add to Cart buttons in our browser. And the following screenshot shows what we see.



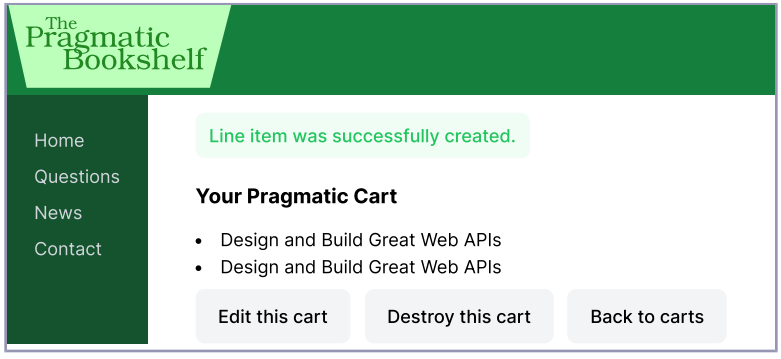
This is a bit underwhelming. We have scaffolding for the cart, but when we created it we didn't provide any attributes, so the view doesn't have anything to show. For now, let's add a trivial template that shows the title of each book in the cart. Update the file views/carts/_cart.html.erb like so:

rails7/depot_f/app/views/carts/_cart.html.erb

```
<div id="<%= dom_id cart %>">
  <h2 class="font-bold text-lg mb-3">Your Pragmatic Cart</h2>

  <ul class="list-disc list-inside">
    <% cart.line_items.each do |item| %>
      <li><%= item.product.title %></li>
    <% end %>
  </ul>
</div>
```

You may be wondering about the underscore in the file name and where the cart variable comes from. Don't worry, we'll cover all this and more when we get to Partial Templates, on page 144, but for now it's enough to know that this is the file that Rails uses to render a single cart.

So, with everything plumbed together, let's go back and click the Add to Cart button again and see our view displayed, as in the next screenshot.



Go back to http://localhost:3000/, the main catalog page, and add a different product to the cart. You'll see the original two entries plus our new item in your cart. It looks like we have sessions working.

We changed the function of our controller, so we know that we need to update the corresponding functional test.

For starters, we only need to pass a product ID on the call to post. Next, we have to deal with the fact that we're no longer redirecting to the line items page. We're instead redirecting to the cart, where the cart ID is internal state data residing in a cookie. Because this is an integration test, instead of focusing on how the code is implemented, we should focus on what users see after following the redirect: a page with a heading identifying that they're looking at a cart, with a list item corresponding to the product they added.

We do this by updating test/controllers/line_items_controller_test.rb:

rails7/depot_g/test/controllers/line_items_controller_test.rb

```
  test "should create line_item" do
    assert_difference("LineItem.count") do
➤     post line_items_url, params: { product_id: products(:ruby).id }
    end

➤   follow_redirect!
➤
➤   assert_select 'h2', 'Your Pragmatic Cart'
➤   assert_select 'li', 'Programming Ruby 1.9'
  end
```

We now rerun this set of tests:

```
depot> bin/rails test test/controllers/line_items_controller_test.rb
```

It's time to show our customer, so we call her over and proudly display our handsome new cart. Somewhat to our dismay, she makes that *tsk-tsk* sound that customers make just before telling you that you clearly don't get something.

Real shopping carts, she explains, don't show separate lines for two of the same product. Instead, they show the product line once with a quantity of 2. It looks like we're lined up for our next iteration.

What We Just Did

It's been a busy, productive day so far. We added a shopping cart to our store, and along the way we dipped our toes into some neat Rails features:

- We created a Cart object in one request and successfully located the same cart in subsequent requests by using a session object.

- We added a private method and placed it in a concern, making it accessible to all of our controllers.
- We created relationships between carts and line items, and relationships between line items and products, and we were able to navigate using these relationships.
- We added a button that causes a product to be posted to a cart, causing a new line item to be created.

Playtime

Here's some stuff to try on your own:

- Add a new variable to the session to record how many times the user has accessed the store controller's index action. Note that the first time this page is accessed, your count won't be in the session. You can test for this with code like this:

  ```
  if session[:counter].nil?
    ...
  ```

  If the session variable isn't there, you need to initialize it. Then you'll be able to increment it.
- Pass this counter to your template, and display it at the top of the catalog page. Hint: the pluralize helper (There's a method to pluralize nouns: on page 382) might be useful for forming the message you display.
- Reset the counter to zero whenever the user adds something to the cart.
- Change the template to display the counter only if the count is greater than five.