In this chapter, you'll see:

- Modifying the schema and existing data
- Error diagnosis and handling
- The flash
- Logging

CHAPTER 10

Task E: A Smarter Cart

Although we have rudimentary cart functionality implemented, we have much to do. To start with, we need to recognize when customers add multiples of the same item to the cart. Once that's done, we'll also have to make sure that the cart can handle error cases and communicate problems encountered along the way to the customer or system administrator, as appropriate.

Iteration E1: Creating a Smarter Cart

Associating a count with each product in our cart is going to require us to modify the line_items table. We've used migrations before; for example, we used a migration in Applying the Migration, on page 72, to update the schema of the database. While that was as part of creating the initial scaffolding for a model, the basic approach is the same:

```
depot> bin/rails generate migration add_quantity_to_line_items quantity:integer
```

Rails can tell from the name of the migration that you're adding columns to the line_items table and can pick up the names and data types for each column from the last argument. The two patterns that Rails matches on are AddXXXToTABLE and RemoveXXXFromTABLE, where the value of XXX is ignored; what matters is the list of column names and types that appears after the migration name.

The only thing Rails can't tell is what a reasonable default is for this column. In many cases, a null value would do, but let's make it the value 1 for existing carts by modifying the migration before we apply it:

rails7/depot_g/db/migrate/20221207000004_add_quantity_to_line_items.rb

```
class AddQuantityToLineItems < ActiveRecord::Migration[7.0]
  def change
➤   add_column :line_items, :quantity, :integer, default: 1
  end
end
```

Once it's complete, we run the migration:

```
depot> bin/rails db:migrate
```

Now we need a smart add_product() method in our Cart, one that checks if our list of items already includes the product we're adding; if it does, it bumps the quantity, and if it doesn't, it builds a new LineItem:

rails7/depot_g/app/models/cart.rb

```
def add_product(product)
  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
```

The find_by() method is a streamlined version of the where() method. Instead of returning an array of results, it returns either an existing LineItem or nil.

We also need to modify the line item controller to use this method:

rails7/depot_g/app/controllers/line_items_controller.rb

```
  def create
    product = Product.find(params[:product_id])
➤   @line_item = @cart.add_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 make two small changes to the _cart template to use this new information:

rails7/depot_g/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-none list-inside">
```

```
    <% cart.line_items.each do |item| %>
➤     <li><%= item.quantity %> &times; <%= item.product.title %></li>
    <% end %>
  </ul>
</div>
```

In addition to displaying the quantity for each line item, we remove the bullets that precede each item in the unordered list by changing list-disc to list-none. This shows one of many benefits to using a CSS framework to make our work more agile. When we make this change, we know not only that this change applies to this particular view; we also know that this change does *not* affect any other view—assurances we don't always have when authoring CSS style sheets.

Now that all the pieces are in place, we can go back to the store page and click the Add to Cart button for a product that's already in the cart. What we're likely to see is a mixture of individual products listed separately and a single product listed with a quantity of two. This is because we added a quantity of one to existing columns instead of collapsing multiple rows when possible. What we need to do next is migrate the data.

We start by creating a migration:

```
depot> bin/rails generate migration combine_items_in_cart
```

This time, Rails can't infer what we're trying to do, so we can't rely on the generated change() method. What we need to do instead is to replace this method with separate up() and down() methods. First, here's the up() method:

rails7/depot_g/db/migrate/20221207000005_combine_items_in_cart.rb

```
def up
  # replace multiple items for a single product in a cart with a
  # single item
  Cart.all.each do |cart|
    # count the number of each product in the cart
    sums = cart.line_items.group(:product_id).sum(:quantity)

    sums.each do |product_id, quantity|
      if quantity > 1
        # remove individual items
        cart.line_items.where(product_id: product_id).delete_all

        # replace with a single item
        item = cart.line_items.build(product_id: product_id)
        item.quantity = quantity
        item.save!
      end
    end
  end
end
```

This is easily the most extensive code you've seen so far. Let's look at it in small pieces:

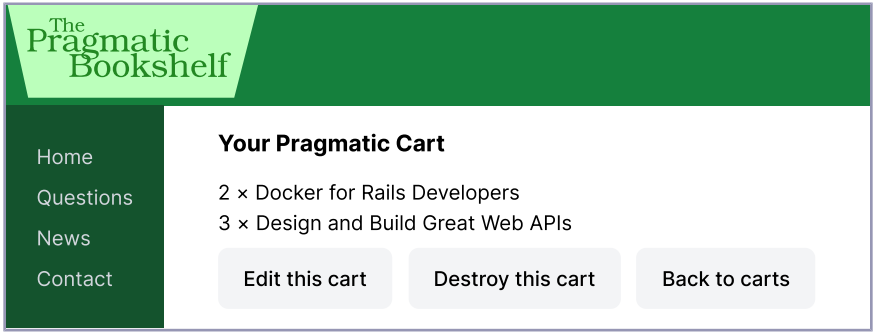
- We start by iterating over each cart.
- For each cart, we get a sum of the quantity fields for each of the line items associated with this cart, grouped by product_id. The resulting sums will be a list of ordered pairs of product_ids and quantity.
- We iterate over these sums, extracting the product_id and quantity from each.
- In cases where the quantity is greater than one, we delete all of the individual line items associated with this cart and this product and replace them with a single line item with the correct quantity.

Note how easily and elegantly Rails enables you to express this algorithm.

With this code in place, we apply this migration like any other migration:

```
depot> bin/rails db:migrate
```

We can see the results by looking at the cart, shown in the following screenshot.



Although we have reason to be pleased with ourselves, we're not done yet. An important principle of migrations is that each step needs to be reversible, so we implement a down() too. This method finds line items with a quantity of greater than one; adds new line items for this cart and product, each with a quantity of one; and, finally, deletes the line item:

rails7/depot_g/db/migrate/20221207000005_combine_items_in_cart.rb

```
def down
  # split items with quantity>1 into multiple items
  LineItem.where("quantity>1").each do |line_item|
    # add individual items
```

```
    line_item.quantity.times do
      LineItem.create(
        cart_id: line_item.cart_id,
        product_id: line_item.product_id,
        quantity: 1
      )
    end

    # remove original item
    line_item.destroy
  end
end
```

Now, we can just as easily roll back our migration with a single command:

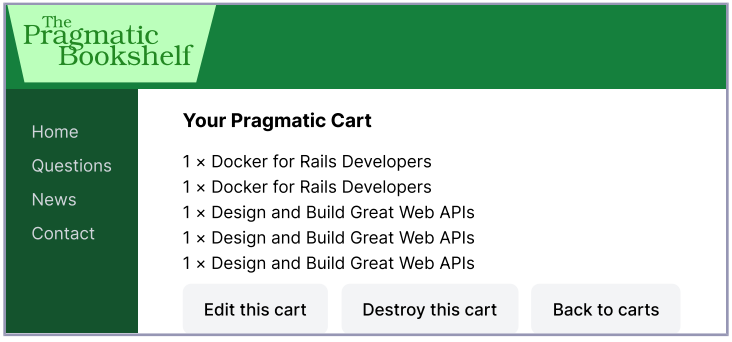
```
depot> bin/rails db:rollback
```

Rails provides a Rake task to allow you to check the status of your migrations:

```
depot> bin/rails db:migrate:status
database: /home/rubys/work/depot/db/development.sqlite3

     Status   Migration ID    Migration Name
    --------------------------------------------------
       up     20160407000001  Create products
       up     20160407000002  Create carts
       up     20160407000003  Create line items
       up     20160407000004  Add quantity to line items
     down     20160407000005  Combine items in cart
```

Now, we can modify and reapply the migration or even delete it entirely. To inspect the results of the rollback, we have to move the migration file out of the way so Rails doesn't think it should apply it. You can do that via `mv`, for example. If you do that, the cart should look like the following screenshot:



Once we move the migration file back and reapply the migration (with the `bin/rails db:migrate` command), we have a cart that maintains a count for each of the products it holds, and we have a view that displays that count.

Since we changed the output the application produces, we need to update the tests to match. Note that what the user sees isn't the string × but the Unicode character ×. If you can't find a way to enter that character using your keyboard and operating system combination, you can use the escape sequence \u00D7[1] instead (also note the use of double quotes, as this is needed in Ruby to enter the escape sequence):

rails7/depot_h/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', "1 \u00D7 Programming Ruby 1.9"
end
```

Happy that we have something presentable, we call our customer over and show her the result of our morning's work. She's pleased—she can see the site starting to come together. However, she's also troubled, having just read an article in the trade press on the way e-commerce sites are being attacked and compromised daily. She read that one kind of attack involves feeding requests with bad parameters into web applications, hoping to expose bugs and security flaws. She noticed that the link to the cart looks like carts/nnn, where nnn is our internal cart ID. Feeling malicious, she manually types this request into a browser, giving it a cart ID of wibble. She's not impressed when our application displays the page shown in the screenshot on page 133.

This seems fairly unprofessional. So our next iteration will be spent making the application more resilient.

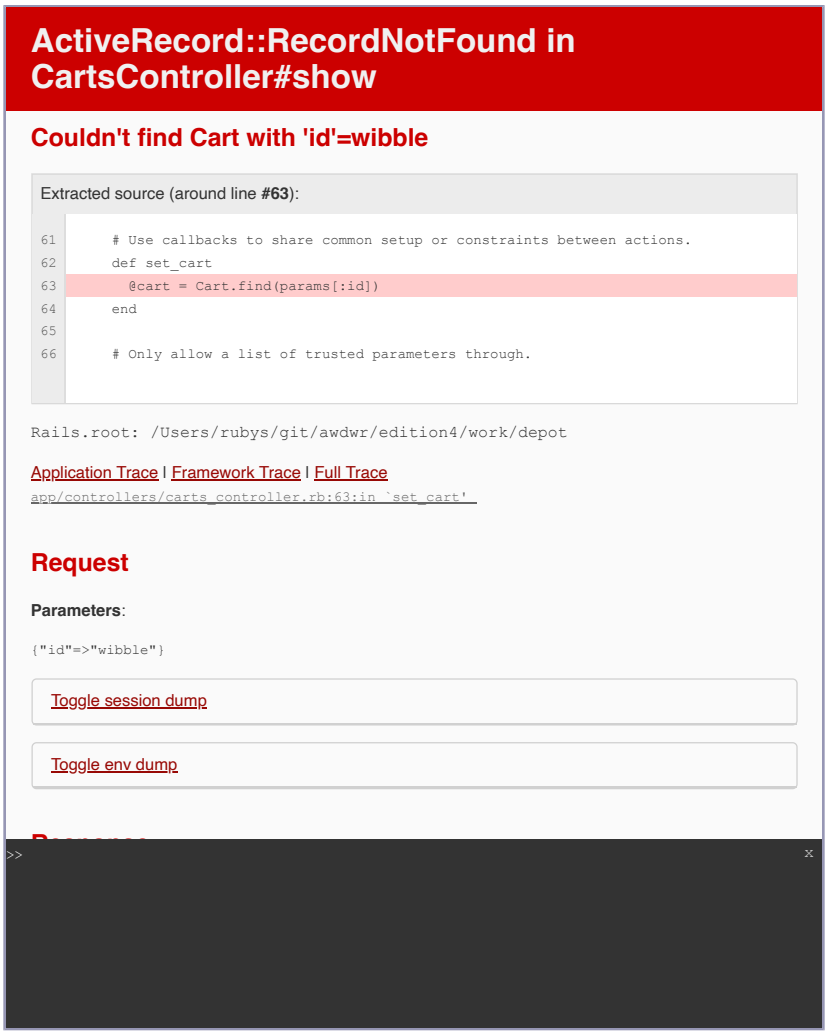
Iteration E2: Handling Errors

It's apparent from the page shown in the screenshot on page 133 that our application raised an exception at line 67 of the carts controller. Your line number might be different, as we have some book-related formatting stuff in our source files. If you go to that line, you'll find the following code:

```
@cart = Cart.find(params[:id])
```

If the cart can't be found, Active Record raises a RecordNotFound exception, which we clearly need to handle. The question arises—how?

1. http://www.fileformat.info/info/unicode/char/00d7/index.htm



We could silently ignore it. From a security standpoint, this is probably the best move, because it gives no information to a potential attacker. However, it also means that if we ever have a bug in our code that generates bad cart IDs, our application will appear to the outside world to be unresponsive—no one will know that an error occurred.

Instead, we'll take two actions when an exception is raised. First, we'll log the fact to an internal log file using the Rails logger facility.[2] Second, we'll redisplay

2. http://guides.rubyonrails.org/debugging_rails_applications.html#the-logger

the catalog page along with a short message (something along the lines of "Invalid cart") to the user, who can then continue to use our site.

Rails has a convenient way of dealing with errors and error reporting. It defines a structure called a *flash*. A flash is a bucket (actually closer to a Hash) in which you can store stuff as you process a request. The contents of the flash are available to the next request in this session before being deleted automatically. Typically, the flash is used to collect error messages. For example, when our show() method detects that it was passed an invalid cart ID, it can store that error message in the flash area and redirect to the index() action to redisplay the catalog. The view for the index action can extract the error and display it at the top of the catalog page. The flash information is accessible within the views via the flash accessor method.

Why can't we store the error in any old instance variable? Remember that after a redirect is sent by our application to the browser, the browser sends a new request back to our application. By the time we receive that request, our application has moved on; all the instance variables from previous requests are long gone. The flash data is stored in the session to make it available between requests.

Armed with this background about flash data, we can create an invalid_cart() method to report on the problem:

rails7/depot_h/app/controllers/carts_controller.rb

```
class CartsController < ApplicationController
  before_action :set_cart, only: %i[ show edit update destroy ]
  rescue_from ActiveRecord::RecordNotFound, with: :invalid_cart
  # GET /carts or /carts.json
  # ...
  private
  # ...

  def invalid_cart
    logger.error "Attempt to access invalid cart #{params[:id]}"
    redirect_to store_index_url, notice: 'Invalid cart'
  end
end
```

The rescue_from clause intercepts the exception raised by Cart.find(). In the handler, we do the following:

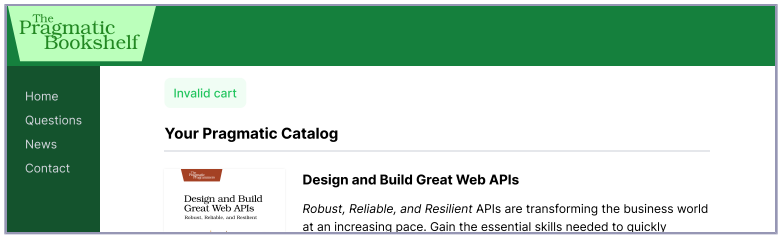
- Use the Rails logger to record the error. Every controller has a logger attribute. Here we use it to record a message at the error logging level.
- Redirect to the catalog display by using the redirect_to() method. The :notice parameter specifies a message to be stored in the flash as a notice. Why

redirect rather than display the catalog here? If we redirect, the user's browser will end up displaying the store URL rather than http://.../cart/wibble. We expose less of the application this way. We also prevent the user from retriggering the error by clicking the Reload button.

With this code in place, we can rerun our customer's problematic query by entering the following URL:

```
http://localhost:3000/carts/wibble
```

We don't see a bunch of errors in the browser now. Instead, the catalog page is displayed with the error message shown in the following screenshot.



If we look at the end of the log file (development.log in the log directory), we see our message:

```
Started GET "/carts/wibble" for 127.0.0.1 at 2016-01-29 09:37:39 -0500
Processing by CartsController#show as HTML
  Parameters: {"id"=>"wibble"}
  ^[[1m^[[35mCart Load (0.1ms)^[[0m  SELECT "carts".* FROM "carts" WHERE
"carts"."id" = ? LIMIT 1  [["id", "wibble"]]
➤ Attempt to access invalid cart wibble
Redirected to http://localhost:3000/
Completed 302 Found in 3ms (ActiveRecord: 0.4ms)
```

On Unix machines, we'd probably use a command such as tail or less to view this file. On Windows, you can use your favorite editor. It's often a good idea to keep a window open to show new lines as they're added to this file. In Unix, you'd use tail -f. You can download a tail command for Windows[3] or get a GUI-based tool.[4] Finally, some OS X users use Console.app to track log files. Just say open development.log at the command line.

This being the Internet, we can't worry only about our published web forms; we have to worry about every possible interface, because malicious crackers can get underneath the HTML we provide and attempt to provide additional

3. http://gnuwin32.sourceforge.net/packages/coreutils.htm
4. http://tailforwin32.sourceforge.net/

parameters. Invalid carts aren't our biggest problem here; we also want to prevent access to *other people's carts*.

As always, your controllers are your first line of defense. Let's go ahead and remove cart_id from the list of parameters that are permitted:

rails7/depot_h/app/controllers/line_items_controller.rb

```
    def line_item_params
➤     params.require(:line_item).permit(:product_id)
    end
```

We can see this in action by rerunning our controller tests:

```
bin/rails test:controllers
```

No tests fail, but a peek into our log/test.log reveals a thwarted attempt to breach security:

```
LineItemsControllerTest: test_should_update_line_item
--------------------------------------------------
  ^[[1m^[[36mLineItem Load (0.0ms)^[[0m  ^[[1m^[[34mSELECT "line_items"
Started PATCH "/line_items/980190962" for 127.0.0.1 at 2022-01-12
Processing by LineItemsController#update as HTML
  Parameters: {"line_item"=>{"cart_id"=>"980190962", "product_id"=>
  ^[[1m^[[36mLineItem Load (0.0ms)^[[0m  ^[[1m^[[34mSELECT "line_items"
➤ ^[[31mUnpermitted parameter: :cart_id. Context: {  }^[[0m
  ^[[1m^[[36mTRANSACTION (0.0ms)^[[0m  ^[[1m^[[35mSAVEPOINT
  ^[[1m^[[36mProduct Load (0.0ms)^[[0m  ^[[1m^[[34mSELECT "products".*
  ^[[1m^[[36mCart Load (0.0ms)^[[0m  ^[[1m^[[34mSELECT "carts".* FROM
  ^[[1m^[[36mTRANSACTION (0.0ms)^[[0m  ^[[1m^[[35mRELEASE SAVEPOINT
Redirected to http://www.example.com/line_items/980190962
Completed 302 Found in 1ms (ActiveRecord: 0.1ms | Allocations: 1669)
  ^[[1m^[[36mTRANSACTION (0.0ms)^[[0m  ^[[1m^[[31mrollback transaction
  ^[[1m^[[36mTRANSACTION (0.0ms)^[[0m  ^[[1m^[[36mbegin transaction
```

Let's clean up that test case to make the problem go away:

rails7/depot_h/test/controllers/line_items_controller_test.rb

```
  test "should update line_item" do
➤     patch line_item_url(@line_item),
➤       params: { line_item: { product_id: @line_item.product_id } }
    assert_redirected_to line_item_url(@line_item)
  end
```

At this point, we clear the test logs and rerun the tests:

```
bin/rails log:clear LOGS=test
bin/rails test:controllers
```

A final scan of the logs identifies no further problems.

It makes good sense to review log files periodically. They hold a lot of useful information.

Sensing the end of an iteration, we call our customer over and show her that the error is now properly handled. She's delighted and continues to play with the application. She notices a minor problem on our new cart display: there's no way to empty items out of a cart. This minor change will be our next iteration. We should make it before heading home.

Iteration E3: Finishing the Cart

We know by now that to implement the empty-cart function, we have to add a link to the cart and modify the destroy() method in the carts controller to clean up the session.

David says:

Battle of the Routes: product_path vs. product_url

It can seem hard in the beginning to know when to use product_path and when to use product_url when you want to link or redirect to a given route. In reality, it's simple.

When you use product_url, you'll get the full enchilada with protocol and domain name, like http://example.com/products/1. That's the thing to use when you're doing redirect_to, because the HTTP spec requires a fully qualified URL when doing 302 Redirect and friends. You also need the full URL if you're redirecting from one domain to another, like product_url(domain: "example2.com", product: product).

The rest of the time, you can happily use product_path. This will generate only the /products/1 part, and that's all you need when doing links or pointing forms, like link_to "My lovely product", product_path(product).

The confusing part is that oftentimes the two are interchangeable because of lenient browsers. You can do a redirect_to with a product_path and it'll probably work, but it won't be valid according to spec. And you can link_to a product_url, but then you're littering up your HTML with needless characters, which is a bad idea too.

Start with the template and use the button_to() method to add a button :

rails7/depot_h/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-none list-inside">
    <% cart.line_items.each do |item| %>
```

```
        <li><%= item.quantity %> &times; <%= item.product.title %></li>
      <% end %>
    </ul>
  </div>

➤ <%= button_to 'Empty Cart', cart, method: :delete,
➤   class: 'ml-4 rounded-lg py-1 px-2 text-white bg-green-600' %>
```

In the controller, let's modify the destroy() method to ensure that the user is deleting his or her own cart (think about it!) and to remove the cart from the session before redirecting to the index page with a notification message:

rails7/depot_h/app/controllers/carts_controller.rb

```
  def destroy
➤   @cart.destroy if @cart.id == session[:cart_id]
➤   session[:cart_id] = nil

    respond_to do |format|
➤     format.html { redirect_to store_index_url,
➤       notice: 'Your cart is currently empty' }
      format.json { head :no_content }
    end
  end
```

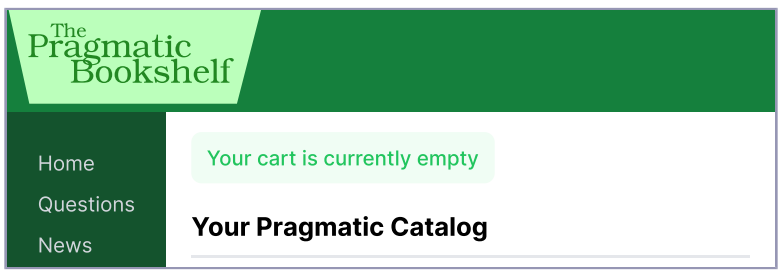
And we update the corresponding test in test/controllers/carts_controller_test.rb:

rails7/depot_i/test/controllers/carts_controller_test.rb

```
  test "should destroy cart" do
➤   post line_items_url, params: { product_id: products(:ruby).id }
➤   @cart = Cart.find(session[:cart_id])
➤
    assert_difference("Cart.count", -1) do
      delete cart_url(@cart)
    end

➤   assert_redirected_to store_index_url
  end
```

Now when we view our cart and click the Empty Cart button, we're taken back to the catalog page and see the message shown in the following screenshot.



We can remove the flash message that's autogenerated when a line item is added:

rails7/depot_i/app/controllers/line_items_controller.rb

```
  def create
    product = Product.find(params[:product_id])
    @line_item = @cart.add_product(product)

    respond_to do |format|
      if @line_item.save
        format.html { redirect_to cart_url(@line_item.cart) }
        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
```

Finally, we get around to tidying up the cart display. The -based approach makes it hard to style. A table-based layout would be easier. Replace app/views/carts/_cart.html.erb with the following:

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

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

  <table class="table-auto">
    <% cart.line_items.each do |line_item| %>
      <tr>
        <td class="text-right"><%= line_item.quantity %></td>
        <td>&times;</td>
        <td class="pr-2">
          <%= line_item.product.title %>
        </td>
        <td class="text-right font-bold">
          <%= number_to_currency(line_item.total_price) %>
        </td>
      </tr>
    <% end %>

    <tfoot>
      <tr>
        <th class="text-right pr-2 pt-2" colspan="3">Total:</th>
        <td class="text-right pt-2 font-bold border-t-2 border-black">
          <%= number_to_currency(cart.total_price) %>
        </td>
      </tr>
```

```
    </tfoot>
  </table>

  <%= button_to 'Empty Cart', cart, method: :delete,
    class: 'ml-4 rounded-lg py-1 px-2 text-white bg-green-600' %>
</div>
```

To make this work, we need to add a method to both the LineItem and Cart models that returns the total price for the individual line item and entire cart, respectively. Here is the line item, which involves only simple multiplication:

rails7/depot_i/app/models/line_item.rb

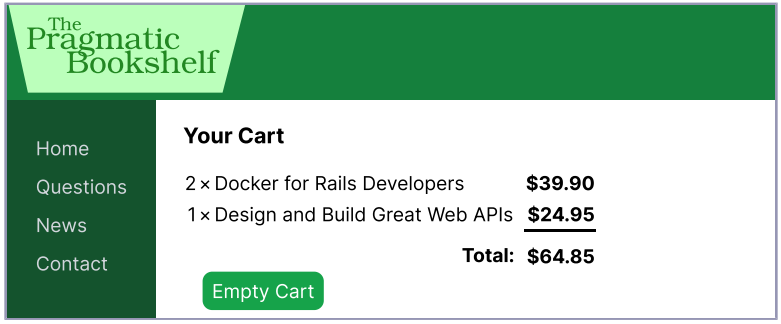
```
def total_price
  product.price * quantity
end
```

We implement the Cart method using the nifty Array::sum() method to sum the prices of each item in the collection:

rails7/depot_i/app/models/cart.rb

```
def total_price
  line_items.sum { |item| item.total_price }
end
```

The following screenshot shows a nicer-looking cart.



Finally, we update our test cases to match the current output:

rails7/depot_i/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 Cart'
    assert_select 'td', "Programming Ruby 1.9"
  end
```

What We Just Did

Our shopping cart is now something the client is happy with. Along the way, we covered the following:

- Adding a column to an existing table, with a default value
- Migrating existing data into the new table format
- Providing a flash notice of an error that was detected
- Using the logger to log events
- Removing a parameter from the permitted list
- Deleting a record
- Adjusting the way a table is rendered, using Tailwind CSS classes

But, just as we think we've wrapped up this functionality, our customer wanders over with a copy of *Information Technology and Golf Weekly*. Apparently, it has an article about the HotWired style of browser interface, where stuff gets updated on the fly. Hmmm…let's look at that tomorrow.

Playtime

Here's some stuff to try on your own:

- Create a migration that copies the product price into the line item, and change the `add_product()` method in the `Cart` model to capture the price whenever a new line item is created. Add prices to the `line_items.yml` fixture.
- Write unit tests that add both unique products and duplicate products to a cart. Assert how many products should be in the cart in each instance. Note that you'll need to modify the fixture to refer to products and carts by name—for example, `product: ruby`.
- Check products and line items for other places where a user-friendly error message would be in order.
- Add the ability to delete individual line items from the cart. This will require buttons on each line, and such buttons will need to be linked to the `destroy()` action in the `LineItemsController`.
- We prevented accessing other users' carts in the `LineItemsController`, but you can still see other carts by navigating directly to a URL like `http://localhost/carts/3`. See if you can prevent accessing any cart other than the one currently stored in the session.