In this chapter, you'll see:

- Linking tables with foreign keys
- Using belongs_to, has_many, and :through
- Creating forms based on models (form_with)
- Linking forms, models, and views
- Generating a feed using atom_helper on model objects

CHAPTER 12

Task G: Check Out!

Let's take stock. So far, we've put together a basic product administration system, we've implemented a catalog, and we have a pretty spiffy-looking shopping cart. So now we need to let the buyer actually purchase the contents of that cart. Let's implement the checkout function.

We're not going to go overboard here. For now, all we'll do is capture the customer's contact information and payment details. Using these, we'll construct an order in the database. Along the way, we'll be looking a bit more at models, validation, and form handling.

Iteration G1: Capturing an Order

An order is a set of line items, along with details of the purchase transaction. Our cart already contains line_items, so all we need to do is add an order_id column to the line_items table and create an orders table based on the Initial guess at application data diagram on page 67, combined with a brief chat with our customer.

First we create the order model and update the line_items table:

```
depot> bin/rails generate scaffold Order name address:text email \
         pay_type:integer
depot> bin/rails generate migration add_order_to_line_item order:references
```

Note that we didn't specify any data type for two of the four columns. This is because the data type defaults to string. This is yet another small way in which Rails makes things easier for you in the most common case without making things any more cumbersome when you need to specify a data type.

Also note that we defined pay_type as an integer. While this is an efficient way to store data that can only store discrete values, storing data in this way requires keeping track of which values are used for which payment type. Rails can do this for you through the use of enum declarations placed in the model class. Add this code to app/models/order.rb:

rails7/depot_o/app/models/order.rb
```
  class Order < ApplicationRecord
➤   enum pay_type: {
      "Check"          => 0,
      "Credit card"    => 1,
      "Purchase order" => 2
    }
  end
```

Finally, we need to modify the second migration to indicate that cart_id can be null in records. This is done by modifying the existing add_reference line to say null: true and adding a new change_column line to enable nulls in the cart_id column.

rails7/depot_o/db/migrate/20221207000008_add_order_to_line_item.rb
```
  class AddOrderToLineItem < ActiveRecord::Migration[7.0]
    def change
➤     add_reference :line_items, :order, null: true, foreign_key: true
➤     change_column :line_items, :cart_id, :integer, null: true
    end
  end
```

Now that we've created the migrations, we can apply them:

```
depot> bin/rails db:migrate
== 20221207000007 CreateOrders: migrating ==============================
-- create_table(:orders)
-> 0.0007s
== 20221207000007 CreateOrders: migrated (0.0022s) =====================

== 20221207000008 AddOrderToLineItem: migrating ========================
-- add_reference(:line_items, :order, {:null=>true, :foreign_key=>true})
   -> 0.0058s
-- change_column(:line_items, :cart_id, :integer, {:null=>true})
   -> 0.0046s
== 20221207000008 AddOrderToLineItem: migrated (0.0246s) ===============
```

Because the database didn't have entries for these two new migrations in the schema_migrations table, the db:migrate task applied both migrations to the database. We could, of course, have applied them separately by running the migration task after creating the individual migrations.

Creating the Order Capture Form

Now that we have our tables and our models as we need them, we can start the checkout process. First, we need to add a Checkout button to the shopping cart. Because it'll create a new order, we'll link it back to a new action in our order controller:

rails7/depot_o/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">
    <%= render cart.line_items %>

    <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>

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

➤     <%= button_to 'Checkout', new_order_path, method: :get,
➤       class: 'ml-4 rounded-lg py-1 px-2 text-black bg-green-200' %>
➤   </div>
</div>
```

We wrapped the buttons in a div and used a flex layout so that they'll appear side by side.

The first thing we want to do is check to make sure that there's something in the cart. This requires us to have access to the cart. Planning ahead, we'll also need this when we create an order:

rails7/depot_o/app/controllers/orders_controller.rb

```
class OrdersController < ApplicationController
➤   include CurrentCart
➤   before_action :set_cart, only: %i[ new create ]
➤   before_action :ensure_cart_isnt_empty, only: %i[ new ]
  before_action :set_order, only: %i[ show edit update destroy ]

  # GET /orders or /orders.json
  #...
➤
➤   private
➤      def ensure_cart_isnt_empty
➤        if @cart.line_items.empty?
➤          redirect_to store_index_url, notice: 'Your cart is empty'
➤        end
➤      end
end
```

If nothing is in the cart, we redirect the user back to the storefront, provide a notice of what we did, and return immediately. This prevents people from navigating directly to the checkout option and creating empty orders. Note that we tucked this handling of an exception case into a before_action method. This enables the main line processing logic to remain clean.

And we add a test for requires item in cart and modify the existing test for should get new to ensure that the cart contains an item:

rails7/depot_o/test/controllers/orders_controller_test.rb

```
➤   test "requires item in cart" do
➤     get new_order_url
➤     assert_redirected_to store_index_path
➤     assert_equal 'Your cart is empty', flash[:notice]
➤   end

    test "should get new" do
➤     post line_items_url, params: { product_id: products(:ruby).id }
➤
      get new_order_url
      assert_response :success
    end
```

Now we want the new action to present users with a form, prompting them to enter the information in the orders table: the user's name, address, email address, and payment type. This means we'll need to display a Rails template containing a form. The input fields on this form will have to link to the corresponding attributes in a Rails model object, so we need to create an empty model object in the new action to give these fields something to work with.

As always with HTML forms, the trick is populating any initial values into the form fields and then extracting those values out into our application when the user clicks the submit button.

In the controller, the order variable is set to reference a new Order model object. This is done because the view populates the form from the data in this object. As it stands, that's not particularly interesting. Because it's a new model object, all the fields will be empty. However, consider the general case. Maybe we want to edit an existing order. Or maybe the user has tried to enter an order but the data has failed validation. In these cases, we want any existing data in the model shown to the user when the form is displayed. Passing in the empty model object at this stage makes all these cases consistent. The view can always assume it has a model object available. Then, when the user clicks the submit button, we'd like the new data from the form to be extracted into a model object back in the controller.

Fortunately, Rails makes this relatively painless. It provides us with a bunch of *form helper* methods. These helpers interact with the controller and with the models to implement an integrated solution for form handling. Before we start on our final form, let's look at a small example:

```
<%= form_with(model: order) do |form| %>
  <p>
    <%= form.label :name, "Name:" %>
    <%= form.text_field :name, size: 40 %>
  </p>
<% end %>
```

This code does two powerful things for us. First, the form_with() helper on the first line sets up an HTML form that knows about Rails routes and models. The argument, model: order, tells the helper which instance variable to use when naming fields and sending the form data back to the controller.

The second powerful feature of the code is how it creates the form fields themselves. You can see that form_with() sets up a Ruby block environment (that ends on the last line of the listing with the end keyword). Within this block, you can put normal template stuff (such as the <p> tag). But you can also use the block's parameter (form in this case) to reference a form context. We use this context to add a text field with a label by calling text_field() and label(), respectively. Because the text field is constructed in the context of form_with, it's automatically associated with the data in the order object. This association means that submitting the form will set the right names and values in the data available to the controller, but it will also pre-populate the form fields with any values already existing on the model.

All these relationships can be confusing. It's important to remember that Rails needs to know both the *names* and the *values* to use for the fields associated with a model. The combination of form_with and the various field-level helpers (such as text_field) gives it this information.

Now we can update the template for the form that captures a customer's details for checkout. It's invoked from the new action in the order controller, so the template is called new.html.erb, found in the app/views/orders directory:

rails7/depot_o/app/views/orders/new.html.erb

```
<div class="mx-auto md:w-2/3 w-full">
➤ <h1 class="font-bold text-4xl">Please Enter Your Details</h1>

  <%= render "form", order: @order %>
</div>
```

In this file, we've updated the h1 and removed the link back to the orders index. This template makes use of a partial named _form. We take a peek at that file and see many long lines repeating the same class definitions. Let's introduce another CSS rule so that we can clean this up:

rails7/depot_o/app/assets/stylesheets/application.tailwind.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

➤ @layer components {
➤   .input-field { @apply
➤      block shadow rounded-md border border-green-400 outline-none
➤      px-3 py-2 mt-2 w-full
➤   }
➤ }
```

There are many reasons to consider factoring out repeated definitions into a style sheet: perhaps it's to reduce repetition to ease maintenance, perhaps it's to reduce visual clutter so that you can focus on the structure of the document, or perhaps it's merely to keep the number of columns down so that it will fit on the printed page. Any of these are good reasons, and they all apply here.

Once we've replaced the class attributes for the form.text_field and wrapped other lines to fit on the page, we make a second set of changes:

rails7/depot_o/app/views/orders/_form.html.erb

```
<%= form_with(model: order, class: "contents") do |form| %>
  <% if order.errors.any? %>
    <div id="error_explanation" class="bg-red-50 text-red-500 px-3 py-2
      font-medium rounded-lg mt-3">
      <h2><%= pluralize(order.errors.count, "error") %>
      prohibited this order from being saved:</h2>

      <ul>
        <% order.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="my-5">
    <%= form.label :name %>
    <%= form.text_field :name, class: "input-field" %>
  </div>

  <div class="my-5">
    <%= form.label :address %>
    <%= form.text_area :address, rows: 4, class: "input-field" %>
  </div>

  <div class="my-5">
    <%= form.label :email %>
➤   <%= form.email_field :email, class: "input-field" %>
  </div>
```

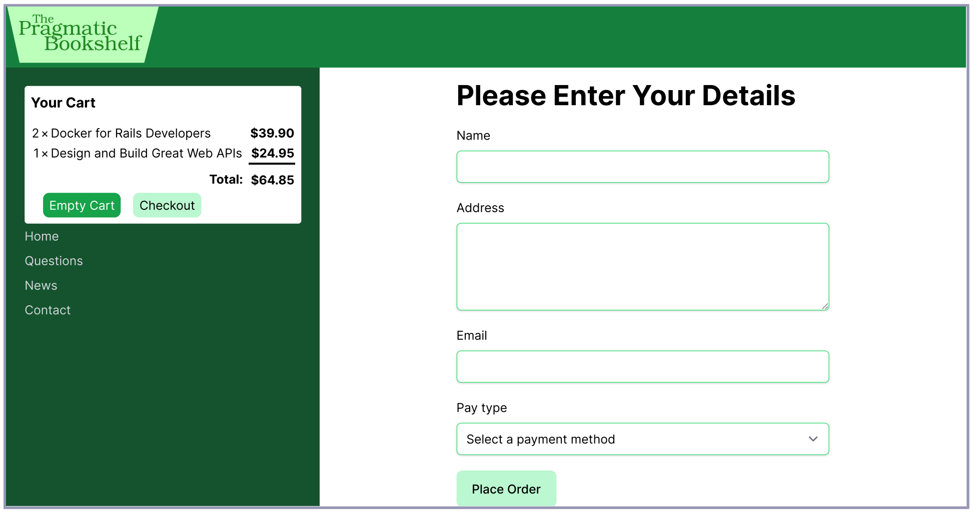
```
    <div class="my-5">
      <%= form.label :pay_type %>
➤     <%= form.select :pay_type, Order.pay_types.keys,
➤                     { prompt: 'Select a payment method' },
➤                     class: "input-field" %>
    </div>

    <div class="inline">
➤     <%= form.submit 'Place Order',  class: "rounded-lg py-3 px-5
➤       bg-green-200 text-black inline-block font-medium cursor-pointer" %>
    </div>
  <% end %>
```

Rails has form helpers for all the different HTML-level form elements. In the preceding code we use text_field, email_field, and text_area helpers to capture the customer's name, email, and address. We'll cover form helpers in more depth in Generating Forms, on page 371.

The only tricky thing in there is the code associated with the selection list. We use the keys defined for the pay_type enum for the list of available payment options. We also pass the :prompt parameter, which adds a dummy selection containing the prompt text.

We also adjust the background and text color of the submit button as well as the text for the button itself.

We're ready to play with our form. Add some stuff to your cart, then click the Checkout button. You should see something like the following screenshot.



Your Cart

2 × Docker for Rails Developers	$39.90
1 × Design and Build Great Web APIs	$24.95
Total:	$64.85

Please Enter Your Details

Name

Address

Email

Pay type

Select a payment method

Looking good! Before we move on, let's finish the new action by adding some validation. We'll change the Order model to verify that the customer enters data for all the input fields. We'll also validate that the payment type is one of the accepted values:

rails7/depot_o/app/models/order.rb

```
class Order < ApplicationRecord
  # ...
➤  validates :name, :address, :email, presence: true
➤  validates :pay_type, inclusion: pay_types.keys
end
```

Some folks might be wondering why we bother to validate the payment type, given that its value comes from a drop-down list that contains only valid values. We do it because an application can't assume that it's being fed values from the forms it creates. Nothing is stopping a malicious user from submitting form data directly to the application, bypassing our form. If the user sets an unknown payment type, that user might conceivably get our products for free.

Note that we already loop over the @order.errors at the top of the page. This'll report validation failures.

Since we modified validation rules, we need to modify our test fixture to match:

rails7/depot_o/test/fixtures/orders.yml

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

one:
➤  name: Dave Thomas
  address: MyText
➤  email: dave@example.org
➤  pay_type: Check

two:
  name: MyString
  address: MyText
  email: MyString
  pay_type: 1
```

Furthermore, for an order to be created, a line item needs to be in the cart, so we need to modify the line items test fixture too:

rails7/depot_o/test/fixtures/line_items.yml

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

one:
  product: two
  cart: one
  price: 1
```

```
two:
➤ product: ruby
➤ order: one
  price: 1
```

Note that if you didn't choose to do the optional exercises in Playtime, on page 141, you need to modify all of the references to products and carts at this time and *not* add price to the line items.

Feel free to make other changes, but only the first is currently used in the functional tests. For these tests to pass, we'll need to implement the model.

Capturing the Order Details

Let's implement the create() action in the controller. This method has to do the following:

1. Capture the values from the form to populate a new Order model object.
2. Add the line items from our cart to that order.
3. Validate and save the order. If this fails, display the appropriate messages, and let the user correct any problems.
4. Once the order is successfully saved, delete the cart, redisplay the catalog page, and display a message confirming that the order has been placed.

We define the relationships themselves, first from the line item to the order:

rails7/depot_o/app/models/line_item.rb

```
class LineItem < ApplicationRecord
➤ belongs_to :order, optional: true
  belongs_to :product
➤ belongs_to :cart, optional: true

  def total_price
    price * quantity
  end
end
```

And then we define the relationship from the order to the line item, once again indicating that all line items that belong to an order are to be destroyed whenever the order is destroyed:

rails7/depot_o/app/models/order.rb

```
class Order < ApplicationRecord
➤ has_many :line_items, dependent: :destroy
  # ...
end
```

The method ends up looking something like this:

rails7/depot_o/app/controllers/orders_controller.rb

```ruby
def create
  @order = Order.new(order_params)
  @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 store_index_url, notice:
        'Thank you for your order.' }
      format.json { render :show, status: :created,
        location: @order }
    else
      format.html { render :new, status: :unprocessable_entity }
      format.json { render json: @order.errors,
        status: :unprocessable_entity }
    end
  end
end
```

We start by creating a new Order object and initialize it from the form data. The next line adds into this order the items that are already stored in the cart; we'll write the method to do that in a minute.

Next, we tell the order object to save itself (and its children, the line items) to the database. Along the way, the order object will perform validation (but we'll get to that in a minute).

If the save succeeds, we do two things. First, we ready ourselves for this customer's next order by deleting the cart from the session. Then we redisplay the catalog, using the redirect_to() method to display a cheerful message. If, instead, the save fails, we redisplay the checkout form with the current cart.

In the create action, we assumed that the order object contains the add_line_items_from_cart() method, so let's implement that method now:

rails7/depot_p/app/models/order.rb

```ruby
class Order < ApplicationRecord
  # ...
  def add_line_items_from_cart(cart)
    cart.line_items.each do |item|
      item.cart_id = nil
      line_items << item
    end
  end
end
```

Joe asks:

Aren't You Creating Duplicate Orders?

Joe is concerned to see our controller creating Order model objects in two actions: new and create. He's wondering why this doesn't lead to duplicate orders in the database.

The answer is that the new action creates an Order object *in memory* simply to give the template code something to work with. Once the response is sent to the browser, that particular object gets abandoned, and it'll eventually be reaped by Ruby's garbage collector. It never gets close to the database.

The create action also creates an Order object, populating it from the form fields. This object *does* get saved in the database. So model objects perform two roles: they map data into and out of the database, but they're also regular objects that hold business data. They affect the database only when you tell them to, typically by calling save().

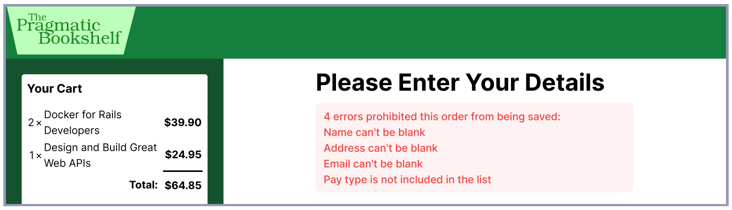
For each item that we transfer from the cart to the order, we need to do two things. First we set the cart_id to nil to prevent the item from going poof when we destroy the cart.

Then we add the item itself to the line_items collection for the order. Notice we didn't have to do anything special with the various foreign-key fields, such as setting the order_id column in the line item rows to reference the newly created order row. Rails does that knitting for us using the has_many() and belongs_to() declarations we added to the Order and LineItem models. Appending each new line item to the line_items collection hands the responsibility for key management over to Rails. We also need to modify the test to reflect the new redirect:

rails7/depot_p/test/controllers/orders_controller_test.rb

```
  test "should create order" do
    assert_difference("Order.count") do
      post orders_url, params: { order: { address: @order.address,
        email: @order.email, name: @order.name,
        pay_type: @order.pay_type } }
    end

➤   assert_redirected_to store_index_url
  end
```

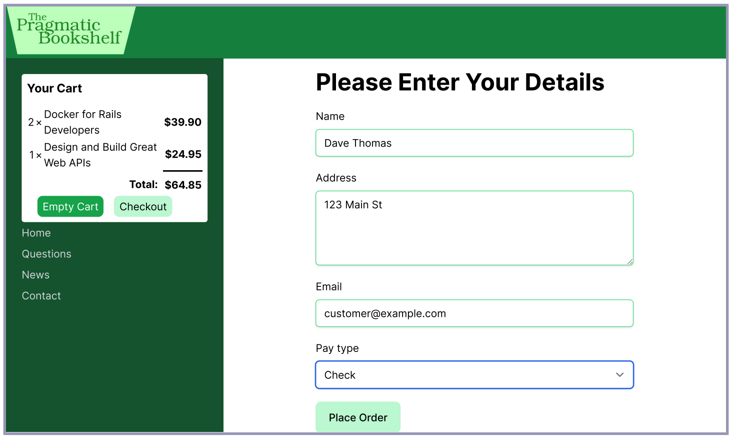
So, as a first test of all of this, click the Place Order button on the checkout page without filling in any of the form fields. You should see the checkout page redisplayed along with error messages complaining about the empty fields, as shown in screenshot on page 176.


The Pragmatic Bookshelf
Your Cart
2 × Docker for Rails Developers $39.90
1 × Design and Build Great Web APIs $24.95
Total: $64.85

Please Enter Your Details

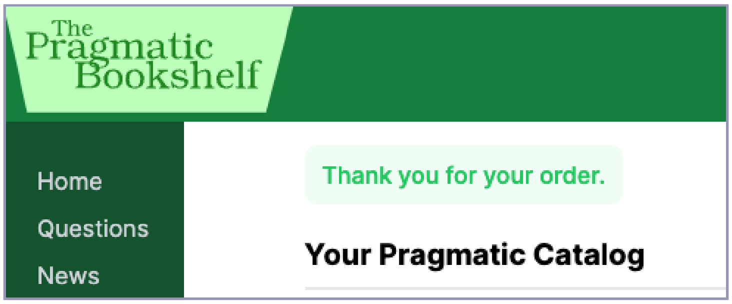
4 errors prohibited this order from being saved:
Name can't be blank
Address can't be blank
Email can't be blank
Pay type is not included in the list

If we fill in data, as shown in the following screenshot, and click Place Order, we should be taken back to the catalog, as shown in next the screenshot.



Please Enter Your Details

Name: Dave Thomas
Address: 123 Main St
Email: customer@example.com
Pay type: Check



Thank you for your order.

Your Pragmatic Catalog

But did it work? Let's look in the database, using the Rails command dbconsole, which tells Rails to open an interactive shell to whatever database we have configured.

```
depot> bin/rails dbconsole
SQLite version 3.36.0 2021-06-18 18:58:49
Enter ".help" for instructions
sqlite> .mode line
sqlite> select * from orders;
               id = 1
             name = Dave Thomas
          address = 123 Main St
            email = customer@example.com
         pay_type = 0
```

```
          created_at = 2022-01-12 16:41:35.897275
          updated_at = 2022-01-12 16:41:48.065263
sqlite> select * from line_items;
                 id = 10
       product_id = 3
          cart_id =
       created_at = 2022-01-12 16:41:46.548932
       updated_at = 2022-01-12 16:41:48.065780
         quantity = 1
            price = 19.95
         order_id = 1
sqlite> .quit
```

Although what you see will differ on details such as version numbers and dates (and price will be present only if you completed the exercises defined in Playtime, on page 141), you should see a single order and one or more line items that match your selections.

Our customer is enthusiastic about our progress, but after playing with the new checkout feature for a few minutes, she has a question: how does a user enter payment details? It's a great question, since there isn't a way to do that. Making that possible is somewhat tricky because each payment method requires different details. If users want to pay with a credit card, they need to enter a card number and expiration date. If they want to pay with a check, we'll need a routing number and an account number. And for purchase orders, we need the purchase order number.

Although we could put all five fields on the screen at once, the customer immediately balks at the poor user experience that would result. Can we show the appropriate fields, depending on what payment type is chosen? Changing elements of a user interface dynamically is certainly possible with some JavaScript but is beyond what we can do with Turbo alone.

Iteration G2: Adding Fields Dynamically to a Form

We need a dynamic form that changes what fields are shown based on what pay type the user has selected. We could cobble something together with jQuery, but Rails includes another framework from the Hotwired set of frameworks that is well suited to this task: Stimulus.[1] Let's put it to use!

Creating a Stimulus Controller

Our starting point is clearly the existing order form. The plan is to add some *additonal fields*, cause those fields to be *hidden* on initial display, and finally,

1. https://stimulus.hotwired.dev/

to expose the fields associated with selected pay type whenever the selection changes.

Let's focus intially on the behavior we want to implement, then on the markup. With Stimulus, the behavior is placed inside a controller, so lets generate one:

```
depot> bin/rails generate stimulus payment
       create  app/javascript/controllers/payment_controller.js
```

What we have is a single file. That's where we place our logic:

rails7/depot_p/app/javascript/controllers/payment_controller.js

```
import { Controller } from "@hotwired/stimulus"

// Connects to data-controller="payment"
export default class extends Controller {
➤   static targets = [ "selection", "additionalFields" ]
➤
➤   initialize() {
➤     this.showAdditionalFields()
➤   }
➤
➤   showAdditionalFields() {
➤     let selection = this.selectionTarget.value
➤
➤     for (let fields of this.additionalFieldsTargets) {
➤       fields.disabled = fields.hidden = (fields.dataset.type != selection)
➤     }
➤   }
}
```

This has three parts:

- First, we declare a list of targets. Targets identify HTML elements that our controller will interact with. Our targets are a selection element and additional fields. We simply list our targets here without specifying how many of each we expect.

- Next, we define the initialization logic, which could implement as a loop over the targets, hiding each, but it turns out that we can take advantage of the code that shows additional fields. This has the additional benefit of gracefully handing the case where the browser restores the value of some form fields when the user manually refreshes the browser window.

- Finally, we define the code that shows the additional fields. We start by getting the value of the selection. We then iterate over the additional fields. Inside the iteration, we either disable and hide each set of fields or enable and show each set based on whether or not the type of those fields matches the selection.

This all sounds straightforward but won't completely make sense until we see the markup. So the next step is to define the additional fields.

Defining Additional Fields

Paying online from your checking account involves providing a routing code and an account number. Let's add these fields to a new partial:

rails7/depot_p/app/views/orders/_check.html.erb

```
<fieldset data-payment-target="additionalFields" data-type="Check">
  <div class="my-5">
    <%= form.label :routing_number %>
    <%= form.text_field :routing_number, class: "input-field" %>
  </div>

  <div class="my-5">
    <%= form.label :account_number %>
    <%= form.password_field :account_number, class: "input-field" %>
  </div>
</fieldset>
```

The first line defines a payment target of additionalFields as well as a type of Check. This matches up with the controller, which defined additionalFields as a target and matches the fields.dataset.type against the value from the selection target.

The remainder of this file is familiar: it defines the two new fields exactly as we have been defining them all along. The only new thing is the reference to a password_field, which causes most browsers to hide the text as you are entering it.

Next up, we need to define fields for a credit card number and an expiration date. We put them into a second partial:

rails7/depot_p/app/views/orders/_cc.html.erb

```
<fieldset data-payment-target="additionalFields" data-type="Credit card">
  <div class="my-5">
    <%= form.label :credit_card_number %>
    <%= form.password_field :credit_card_number, class: "input-field" %>
  </div>

  <div class="my-5">
    <%= form.label :expiration_date %>
    <%= form.text_field :expiration_date, class: "input-field",
      size:9, placeholder: "e.g. 03/22" %>
  </div>
</fieldset>
```

No surprises here. Finally, we need a purchase order number field, which we put into a third partial:

rails7/depot_p/app/views/orders/_po.html.erb

```
<fieldset data-payment-target="additionalFields" data-type="Purchase order">
  <div class="my-5">
    <%= form.label :po_number %>
    <%= form.number_field :po_number, class: "input-field" %>
  </div>
</fieldset>
```

Now that we're done with the additional fields, it's time to update the form itself:

rails7/depot_p/app/views/orders/_form.html.erb

```
<%= form_with(model: order, class: "contents") do |form| %>
  <% if order.errors.any? %>
    <div id="error_explanation" class="bg-red-50 text-red-500 px-3 py-2
      font-medium rounded-lg mt-3">
      <h2><%= pluralize(order.errors.count, "error") %>
      prohibited this order from being saved:</h2>

      <ul>
        <% order.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="my-5">
    <%= form.label :name %>
    <%= form.text_field :name, class: "input-field" %>
  </div>

  <div class="my-5">
    <%= form.label :address %>
    <%= form.text_area :address, rows: 4, class: "input-field" %>
  </div>

  <div class="my-5">
    <%= form.label :email %>
    <%= form.email_field :email, class: "input-field" %>
  </div>

➤ <div data-controller="payment">
    <div class="my-5">
      <%= form.label :pay_type %>
      <%= form.select :pay_type, Order.pay_types.keys,
                      { prompt: 'Select a payment method' },
➤                     'data-payment-target' => 'selection',
➤                     'data-action' => 'payment#showAdditionalFields',
                      class: "input-field" %>
    </div>
➤
➤   <%= render partial: 'check', locals: {form: form} %>
➤     <%= render partial: 'cc', locals: {form: form} %>
➤     <%= render partial: 'po', locals: {form: form} %>
➤   </div>

  <div class="inline">
    <%= form.submit 'Place Order',  class: "rounded-lg py-3 px-5
      bg-green-200 text-black inline-block font-medium cursor-pointer" %>
  </div>
<% end %>
```

This file has three sets of changes.

- First, we wrap all of the elements that are to be controlled by the payment Stimulus controller with a div element containing a data-controller field naming the controller.
- Next, we identify the form.select element as the selection target for the payment controller and associate an action by naming the method to be called when the selection changes.
- Finally, we render the three partials that we just created.

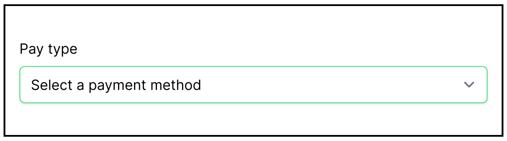
With both the code and markup now in place, we revisit the browser to see the results shown in the screenshots on page 182.

If that isn't what you're seeing, here are some things to check:

- Your browser's console is always a great resource and where you'll find both syntax and runtime errors in your JavaScript code.
- Check for typos in your markup and in the portions of the payment Stimulus controller that need to match your markup. Remember that generally the default is to do nothing. If the controller doesn't match, then no code will be executed. If no additional fields are found, the loop will not hide anything.
- Feel free to add calls to console.log inside your Stimulus controller.

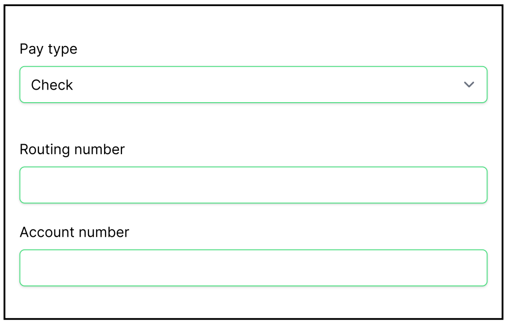
Now that users can check out and purchase products, the customer needs a way to view these orders. Going into the database directly isn't acceptable. We also don't have time to build a full-fledged admin user interface right now, so we'll take advantage of the various Atom feed readers that exist and have our app export all the orders as an Atom feed so the customer can quickly see what's been purchased.

For the times when you really want to run all of your tests with a single command, Rails has this covered too: try running bin/rails test:all.



Pay type

Select a payment method

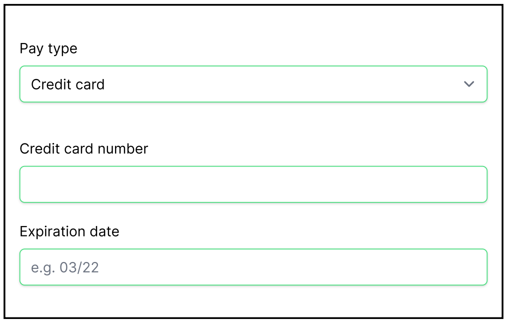


Pay type

Check

Routing number

Account number



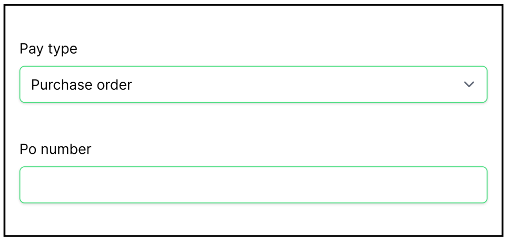
Pay type

Credit card

Credit card number

Expiration date

e.g. 03/22



Pay type

Purchase order

Po number

What We Just Did

In a fairly short amount of time, we did the following:

- We created a form to capture details for the order and linked it to a new order model.
- We added validation and used helper methods to display errors to the user.
- We provided a feed so the administrator can monitor incoming orders.

Playtime

Here's some stuff to try on your own:

- Get HTML- and JSON-formatted views working for `who_bought` requests. Experiment with including the order information in the JSON view by rendering `@product.to_json(include: :orders)`. Do the same thing for XML using `ActiveModel::Serializers::Xml`.[2]

- What happens if you click the Checkout button in the sidebar while the checkout screen is already displayed? Can you find a way to disable the button in this circumstance?

- The list of possible payment types is currently stored as a constant in the `Order` class. Can you move this list into a database table? Can you still make validation work for the field?

Iteration G3: Testing Our JavaScript Functionality

Now that we have application-level functionality in JavaScript code, we're going to need to have tests in place to ensure that the function not only works as intended but continues to work as we make changes to the application.

Testing this functionality involves a lot of steps: visiting the store, selecting an item, adding that item to the cart, clicking checkout, filling in a few fields, and selecting a payment type. And from a testing perspective, we're going to need both a Rails server and a browser.

To accomplish this, Rails makes use of the popular Google Chrome web browser and Capybara,[3] which is a tool that drives this automation. Microsoft Edge and Mozilla's Firefox are also supported, as is Apple's Safari once Allow Remote Automation is enabled via the Develop menu.

Tests that pull together a complete and integrated version of the software are called *system tests*, and that's exactly what we'll be doing: we'll be testing a full end-to-end scenario with a web browser, web server, our application, and a database.

When we created scaffolds in previous chapters, Rails created system tests for us that performed basic checks. Let's run those now to make sure they're passing and that system testing is working. If you're using a virtual machine, you'll need to make one change before running the

2. https://github.com/rails/activemodel-serializers-xml#readme
3. https://github.com/teamcapybara/capybara#readme

tests. Edit test/application_system_test_case.rb and change :chrome to :headless_chrome so that the system tests use a browser that doesn't need to pop up on the screen, like so:

rails7/depot_p/test/application_system_test_case.rb
```
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
➤   driven_by :selenium, using: :headless_chrome, screen_size: [1400, 1400]
end
```

Should you wish to test with a different browser, this is the place where you would indicate which browser to use. :edge, :firefox, and :safari are all supported.

Let's run the existing system tests using bin/rails test:system. Oh dear, there are about a dozen failures—which isn't all that surprising, given that we've ignored these tests up to this point.

The output indicates screenshot images have been placed into the /tmp/screenshots directory, and taking a look at a few of them, we feel a bit like archaeologists. The tests verify the operation of the code as originally scaffolded—most importantly before we added product validation logic in Iteration B1: Validating!, on page 85, and before we moved the cart in Changing the Flow, on page 147.

We could fix these errors, but we would end up with tests that largely duplicate tests we already have. Lets clean things up and write an entirely new test—one that takes advantage of the fact that we're interacting with a real browser that runs the JavaScript code that we provided.

```
$ rm test/system/carts_test.rb
$ rm test/system/line_items_test.rb
$ rm test/system/product_test.rb
```

Now we're ready to write the test we came here to write, which is that our JavaScript is working when it's run in a web browser. We start by describing the actions and checks we want performed in test/system/orders_test.rb, which already has some tests in it from the scaffold:

rails7/depot_p/test/system/orders_test.rb
```
require "application_system_test_case"

class OrdersTest < ApplicationSystemTestCase
➤   test "check dynamic fields" do
➤     visit store_index_url
➤
➤     click_on 'Add to Cart', match: :first
➤
➤     click_on 'Checkout'
➤
```

```
➤     assert has_no_field? 'Routing number'
➤     assert has_no_field? 'Account number'
➤     assert has_no_field? 'Credit card number'
➤     assert has_no_field? 'Expiration date'
➤     assert has_no_field? 'Po number'
➤
➤     select 'Check', from: 'Pay type'
➤
➤     assert has_field? 'Routing number'
➤     assert has_field? 'Account number'
➤     assert has_no_field? 'Credit card number'
➤     assert has_no_field? 'Expiration date'
➤     assert has_no_field? 'Po number'
➤
➤     select 'Credit card', from: 'Pay type'
➤
➤     assert has_no_field? 'Routing number'
➤     assert has_no_field? 'Account number'
➤     assert has_field? 'Credit card number'
➤     assert has_field? 'Expiration date'
➤     assert has_no_field? 'Po number'
➤
➤     select 'Purchase order', from: 'Pay type'
➤
➤     assert has_no_field? 'Routing number'
➤     assert has_no_field? 'Account number'
➤     assert has_no_field? 'Credit card number'
➤     assert has_no_field? 'Expiration date'
➤     assert has_field? 'Po number'
➤   end
  end
```

As you can see, it's largely a repetition of a few lines of code with minor variations, prefaced by a few discrete steps: visit() a URL, find the :first button with the text "Add to Cart" and click_on() it. Then click_on() the button labeled "Checkout". We then select() various pay types and verify what fields we expect to see and what fields we expect *not* to see.

At this point in the test, we check an assumption that the routing number field is not on the page yet. We do this using has_no_field?() and pass it "Routing number", which is a the text the user would see if they had selected Check as the Pay type. We repeat this for all the other fields that the user could eventually see but at this point should be hidden.

In general, be careful when using has_no_field?() as there are an uncountable number of fields the form doesn't have, and any typo will cause such a test to pass. In this case we're safe, as the test contains matching has_field?() method calls.

After that, we select() the value "Check" from the "Pay type" selector and then assert that the routing number text field showed up, using has_field(). We repeat this for each combination of Pay type and field. Four groups of five assertions, for a total of twenty asssertions. Whew!

Capybara makes all of this possible using a compact, readable API that requires very little code. For additional information and more methods, we suggest that you familiarize yourself with the domain-specific language (DSL) that Capybara provides.[4]

Now let's run the test we just wrote:

```
$ bin/rails test:system
Running 5 tests in a single process (parallelization threshold is 50)
Run options: --seed 55897

# Running:

Capybara starting Puma...
* Version 5.5.2 , codename: Zawgyi
* Min threads: 0, max threads: 4
* Listening on tcp://127.0.0.1:56776
Capybara starting Puma...
* Version 3.12.1 , codename: Llamas in Pajamas
* Min threads: 0, max threads: 4
* Listening on tcp://127.0.0.1:43749
.....

Finished in 4.065668s, 1.2298 runs/s, 5.9031 assertions/s.

5 runs, 24 assertions, 0 failures, 0 errors, 0 skips
```

When you run this, you'll notice a number of things. First, a web server is started on your behalf, and then a browser is launched and the actions you requested are performed. Once the test is complete, both are stopped and the results of the test are reported back to you. All this is based on your instructions as to what actions and tests are to be performed, and it's then expressed clearly and succinctly as a system test.

Note that system tests tend to take a bit longer to execute than model or controller tests, which is why they're not run as a part of bin/rails test. But all in all, these tests aren't all that slow, and they can test things that can't be tested in any other way, so system tests are a valuable tool to have in our toolchest.

4. https://github.com/teamcapybara/capybara#the-dsl

What We Just Did

- We replaced a static form_select field with a dynamic list of form fields that change instantly based on user selection.
- We wrote a Stimulus controller that attached to the HTML to make the dynamic changes happen.
- We used Capybara to system-test this functionality.

Playtime

Here's some stuff to try on your own:

- Add an order and check the logs, and you'll see a number of *Unpermitted parameters* messages. While new fields were added to the form, they have yet to be added to the database. Generate a migration to add the fields, and add them to the order_parameters() method.
- Add a test to verify that the Add to Cart and Empty Cart buttons reveal and hide the cart, respectively.
- Add a test of the highlight feature you added in Iteration F3: Highlighting Changes, on page 155. The Capybara have_css() method[5] may be useful here.

5. https://rubydoc.info/github/jnicklas/capybara/Capybara%2FRSpecMatchers:have_css