Extracted from:

Agile Web Development with Rails 4

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails 4

Agile Web Development with Rails 4

Sam Ruby
Dave Thomas
David Heinemeier Hansson

Edited by Susannah Davidson Pfalzer

RAILS

The Facets of Ruby Series

Agile Web Development with Rails 4

Sam Ruby
Dave Thomas
David Heinemeier Hansson

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Susannah Pfalzer (editor)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-56-7
Printed on acid-free paper.
Book version: P1.0—September 2013

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 between models, and adding a button to the view, so let's get started.

9.1 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 cart.

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 the identity from the session and use it to find the cart in the database.

Let's go ahead and create a cart.

```
depot> rails generate scaffold Cart
 ...
depot> rake 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 symbol :cart_id.

Download rails40/depot_f/app/controllers/concerns/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 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 is not found (which will happen if the ID is nil or invalid for any reason), then this method will proceed to create a new Cart, store the ID of the created cart into the session, and then return the new cart.

rescue
↪ on page ?

Note that we place the set_cart() method in a CurrentCart module and mark it as private. This treatment allows us to share common code (even as little as a single method!) between controllers and furthermore prevents Rails from ever making it available as an action on the controller.

private
↪ on page ?

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

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

Download rails40/depot_f/app/models/line_item.rb

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :cart
end
```

At the model level, there is no difference between a simple reference and a "belongs to" relationship. Both are implemented using the belongs_to() method. In the LineItem model, the two belongs_to() calls tell Rails that rows in the line_items table are children of rows in the carts and products tables. No line item can exist unless the corresponding cart and product rows exist. There's an easy way to remember where to put belongs_to declarations: if a table has foreign keys, the corresponding model should have a belongs_to for each.

Just 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().

Download rails40/depot_f/app/models/cart.rb

```
  class Cart < ActiveRecord::Base
➤   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'll 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 will make use of validation code to prevent the removal of products that are referenced by line items.

Download rails40/depot_f/app/models/product.rb

```
  class Product < ActiveRecord::Base
➤   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
```

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 returns false, the row will not be destroyed.

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

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

9.3 Iteration D3: Adding a Button

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

There is no need to create a new controller or even a new action. Taking a look at the actions provided by the scaffold generator, you 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 is 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 are 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 will add a button this time; this means we will 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 will 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. That's easy enough—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:

Download rails40/depot_f/app/views/store/index.html.erb

```
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>

<h1>Your Pragmatic Catalog</h1>

<% cache ['store', Product.latest] do %>
  <% @products.each do |product| %>
    <% cache ['entry', product] do %>
      <div class="entry">
        <%= image_tag(product.image_url) %>
        <h3><%= product.title %></h3>
        <%= sanitize(product.description) %>
        <div class="price_line">
          <span class="price"><%= number_to_currency(product.price) %></span>
➤         <%= button_to 'Add to Cart', line_items_path(product_id: product) %>
        </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

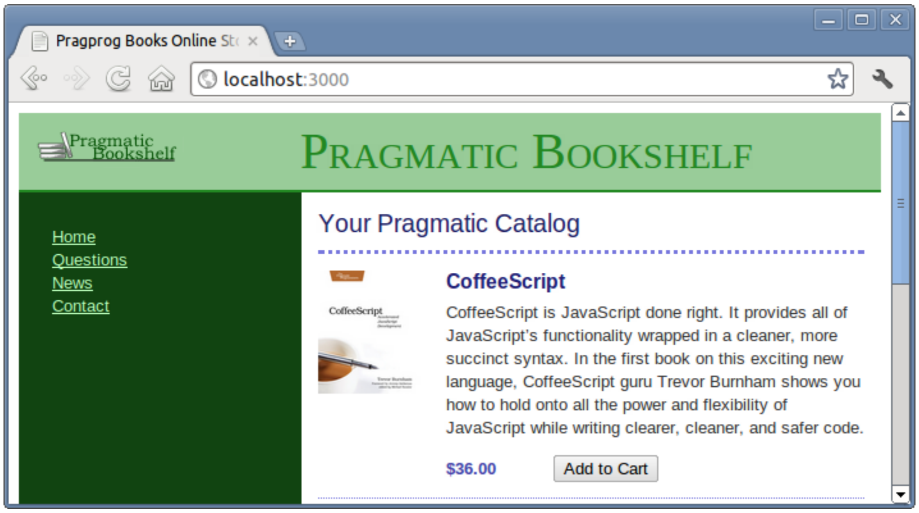
There's one more formatting issue. button_to creates an HTML <form>, and that form contains an HTML <div>. Both of these are normally block elements, which will appear on the next line. We'd like to place them next to the price, so we need to add a little CSS magic to make them inline.

Download rails40/depot_f/app/assets/stylesheets/store.css.scss

```
p, div.price_line {
  margin-left: 100px;
  margin-top: 0.5em;
  margin-bottom: 0.8em;

➤ form, div {
➤   display: inline;
➤ }
}
```

Now our index page looks like the following figure. 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.



Your Pragmatic Catalog

CoffeeScript

CoffeeScript is JavaScript done right. It provides all of JavaScript's functionality wrapped in a cleaner, more succinct syntax. In the first book on this exciting new language, CoffeeScript guru Trevor Burnham shows you how to hold onto all the power and flexibility of JavaScript while writing clearer, cleaner, and safer code.

$36.00

Figure 20—Now there's an Add to Cart button!

Why the create() method? The default HTTP method for a link is a get, the default HTTP method for a button is a post, and 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 there isn't one 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 5 to find (or create) a cart in the session.

Download rails40/depot_f/app/controllers/line_items_controller.rb
```
class LineItemsController < ApplicationController
➤ include CurrentCart
➤ before_action :set_cart, only: [:create]
  before_action :set_line_item, only: [:show, :edit, :update, :destroy]

  # GET /line_items
  #...
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 ?, but for now all we 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.[1] to build the line item itself.

Download rails40/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 @line_item.cart,
        notice: 'Line item was successfully created.' }
      format.json { render action: 'show',
        status: :created, location: @line_item }
    else
      format.html { render action: 'new' }
      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 is 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 will take 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 will cover in more detail in Section 10.2, *Iteration E2: Handling Errors*, on page ?, and handling JSON requests. But for now, we want to modify only one

1. Some lines have been wrapped to fit on the page.

more thing: once the line item is created, we want to redirect you 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.

As we changed the function of our controller, we know that we will need to update the corresponding functional test. We need to pass a product ID on the call to create and change what we expect for the target of the redirect. We do this by updating test/controllers/line_items_controller_test.rb.

Download rails40/depot_g/test/controllers/line_items_controller_test.rb

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

➤ assert_redirected_to cart_path(assigns(:line_item).cart)
end
```

While we haven't talked about the assigns method to date, we have already been using it because it is generated automatically by the scaffold command. This method gives us access to the instance variables that have been (or can be) assigned by controller actions for use in views.

We now rerun this set of tests.

```
depot> rake test test/controllers/line_items_controller_test.rb
```

Confident that the code works as intended, we try the Add to Cart buttons in our browser.

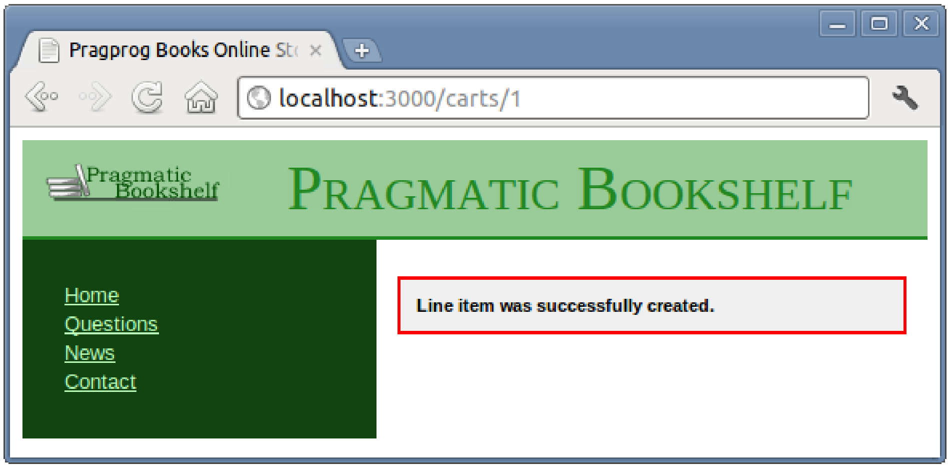
And Figure 21, *Confirmation that the request was processed*, on page 13 shows what we see.

This is a bit underwhelming. Although we have scaffolding for the cart, when we created it, we didn't provide any attributes, so the view doesn't have anything to show. For now, let's write a trivial template (we'll tart it up in a minute).

Download rails40/depot_f/app/views/carts/show.html.erb

```
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>

<h2>Your Pragmatic Cart</h2>
<ul>
  <% @cart.line_items.each do |item| %>
    <li><%= item.product.title %></li>
  <% end %>
</ul>
```



Line item was successfully created.

Figure 21—Confirmation that the request was processed

So, with everything plumbed together, let's go back and click the Add to Cart button again and see our simple view displayed, as in Figure 22, *Cart with new item displayed*, on page 14.

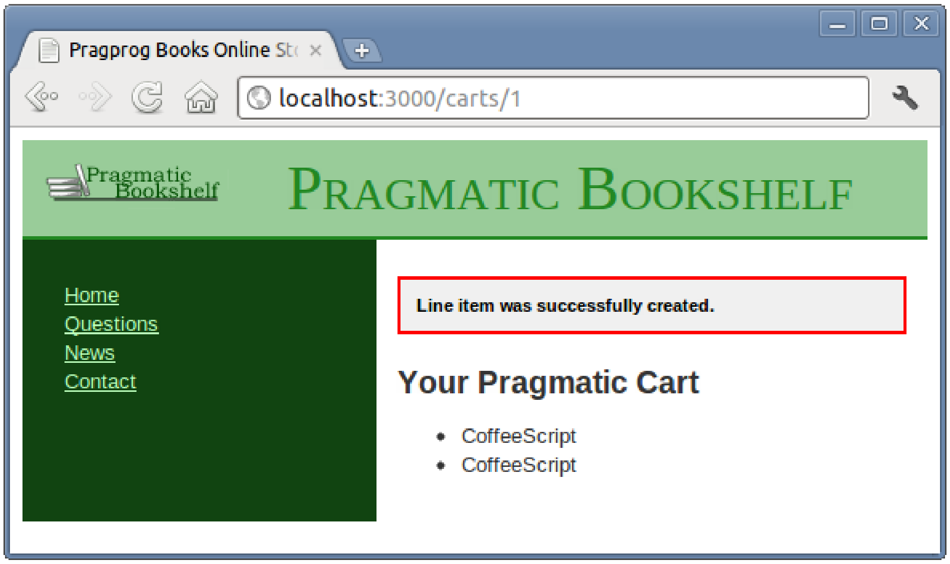
Go back to http://localhost:3000/, the main catalog page, and add a different product to the cart. You'll see the original two entries plus our new item in your cart. It looks like we have sessions working. It's time to show our customer, so we call her over and proudly display our handsome new cart. 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 has been a busy, productive day so far. We've added a shopping cart to our store, and along the way we've dipped our toes into some neat Rails features.

- We created a Cart object in one request and were able to successfully locate the same cart in subsequent requests using a session object.
- We added a private method and placed it in a concern, making it accessible to all of our controllers.



Line item was successfully created.

Your Pragmatic Cart

- CoffeeScript
- CoffeeScript

Figure 22—Cart with new item displayed

- 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 caused 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'll 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 (definition on page ?) might be useful when 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 it is greater than five.

(You’ll find hints at http://pragprog.com/wikis/wiki/RailsPlayTime.)