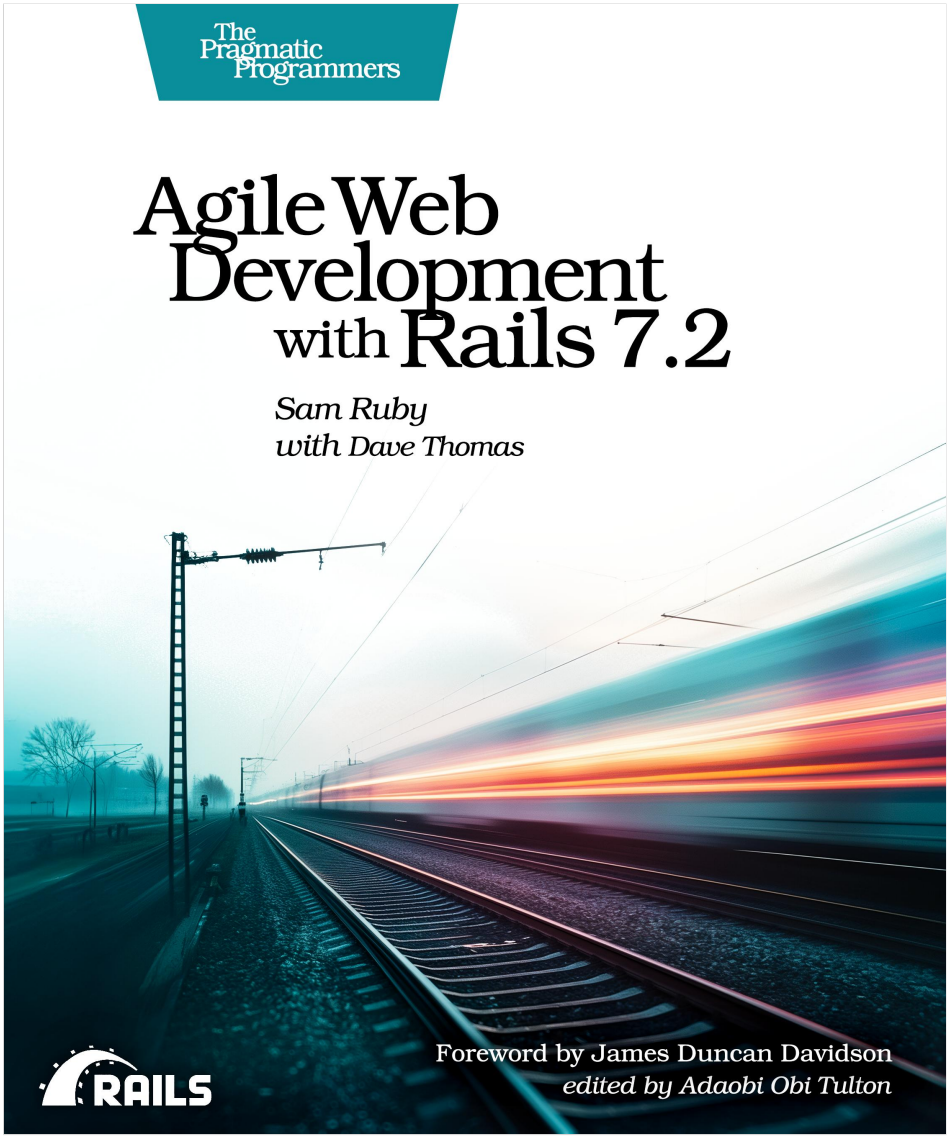



This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

## Iteration C1: Creating the Catalog Listing

We've already created the products controller, used by the seller to administer the Depot application. Now it's time to create a second controller, one that interacts with the paying customers. Let's call it Store:

```
depot> bin/rails generate controller Store index
   create  app/controllers/store_controller.rb
    route  get 'store/index'
   invoke  tailwindcss
   create    app/views/store
   create    app/views/store/index.html.erb
   invoke  test_unit
   create    test/controllers/store_controller_test.rb
   invoke  helper
   create    app/helpers/store_helper.rb
   invoke    test_unit
```

As in the previous chapter, where we used the generate utility to create a controller and associated scaffolding to administer the products, here we've asked it to create a controller (the StoreController class in the store_controller.rb file) containing a single action method, index().

While everything is already set up for this action to be accessed via http://localhost:3000/store/index (feel free to try it!), we can do better. Let's simplify things and make this the root URL for the website. We do this by editing config/routes.rb:

rails72/depot_d/config/routes.rb

```
Rails.application.routes.draw do
  root "store#index", as: "store_index"
  resources :products
  # Define your application routes per the DSL in
  # https://guides.rubyonrails.org/routing.html

  # Reveal health status on /up that returns 200 if the app boots with no
  # exceptions, otherwise 500.
  # Can be used by load balancers and uptime monitors to verify that the
  # app is live.
  get "up" => "rails/health#show", as: :rails_health_check

  # Render dynamic PWA files from app/views/pwa/*
  get "service-worker" => "rails/pwa#service_worker", as: :pwa_service_worker
  get "manifest" => "rails/pwa#manifest", as: :pwa_manifest

  # Defines the root path route ("/")
  # root "posts#index"
end
```

We've replaced the get 'store/index' line with a call to define a root path, and in the process we added an as: 'store_index' option. The latter tells Rails to create

store_index_path and store_index_url accessor methods, enabling existing code—and tests!—to continue to work correctly. Let's try it. Point a browser at http://localhost:3000/, and up pops our web page. See the following screenshot.


It might not make us rich, but at least we know everything is wired together correctly. It even tells us where to find the template file that draws this page.

Let's start by displaying a list of all the products in our database. We know that eventually we'll have to be more sophisticated, breaking them into categories, but this'll get us going.

We need to get the list of products out of the database and make it available to the code in the view that'll display the table. This means we have to change the index() method in store_controller.rb. We want to program at a decent level of abstraction, so let's assume we can ask the model for a list of the products:

rails72/depot_d/app/controllers/store_controller.rb

```
class StoreController < ApplicationController
  def index
➤   @products = Product.order(:title)
  end
end
```

We asked our customer if she had a preference regarding the order things should be listed in, and we jointly decided to see what happens if we display the products in alphabetical order. We do this by adding an order(:title) call to the Product model.

Now we need to write our view template. To do this, edit the index.html.erb file in app/views/store. (Remember that the path name to the view is built from the name of the controller [store] and the name of the action [index]. The .html.erb part signifies an ERB template that produces an HTML result.)

rails72/depot_d/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>
```

```
  <% @products.each do |product| %>
    <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">
          <%= product.price %>
        </div>
      </div>
    </li>
  <% end %>
</ul>
</div>
```

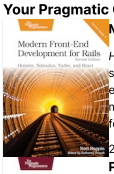
Note the use of the sanitize() method for the description. This allows us to safely[1] add HTML stylings to make the descriptions more interesting for our customers.

We also used the image_tag() helper method. This generates an HTML <img> tag using its argument as the image source.

A page refresh brings up the display shown in the following screenshot. It's still pretty basic, and it seems to be missing something. The customer happens to be walking by as we ponder this, and she points out that she'd also like to see a decent-looking banner and sidebar on public-facing pages.

1. https://owasp.org/www-community/attacks/xss/

**Your Pragmatic Catalog**




**Modern Front-End Development for Rails, Second Edition**

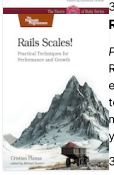
*Hotwire, Stimulus, Turbo, and React* Improve the user experience for your Rails app with rich, engaging client-side interactions. Learn to use the Rails 7 tools and simplify the complex JavaScript ecosystem. It's easier than ever to build user interactions with Hotwire, Turbo, and Stimulus. You can add great front-end flair without much extra complication. Use React to build a more complex set of client-side features. Structure your code for different levels of client-side needs with these powerful options. Add to your toolkit today!

28.95


**Programming Ruby 3.3 (5th Edition)**

*The Pragmatic Programmers' Guide* Ruby is one of the most important programming languages in use for web development. It powers the Rails framework, which is the backing of some of the most important sites on the web. The Pickaxe Book, named for the tool on the cover, is the definitive reference on Ruby, a highly-regarded, fully object-oriented programming language. This updated edition is a comprehensive reference on the language itself, with a tutorial on the most important features of Ruby—including pattern matching and Ractors—and describes the language through Ruby 3.3.

33.95




**Rails Scales!**

*Practical Techniques for Performance and Growth* Rails doesn't scale. So say the naysayers. They're wrong. Ruby on Rails runs some of the biggest sites in the world, impacting the lives of millions of users while efficiently crunching petabytes of data. This book reveals how they do it, and how you can apply the same techniques to your applications. Optimize everything necessary to make an application function at scale: monitoring, product design, Ruby code, software architecture, database access, caching, and more. Even if your app may never have millions of users, you reduce the costs of hosting and maintaining it.

30.95

At this point in the real world, we'd probably want to call in the design folks. But Pragmatic Web Designer is off getting inspiration on a beach somewhere and won't be back until later in the year, so let's put a placeholder in for now. It's time for another iteration.