

Extracted from:

# Agile Web Development with Rails

---

Third Edition

This PDF file contains pages extracted from Agile Web Development with Rails, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

**Note:** This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2009 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The  
Pragmatic  
Programmers

Covers  
Rails 2

# Agile Web Development with Rails

Third Edition



*Sam Ruby  
Dave Thomas*

*David Heinemeier Hansson*

*with Leon Breedt, Mike Clark, Justin Gehtland,  
James Duncan Davidson, and Andreas Schurz*



The Facets  of Ruby Series



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragprog.com>

Copyright © 2009 The Pragmatic Programmers LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-16-6

ISBN-13: 978-1-9343561-6-6

Printed on acid-free paper.

P2.0 printing, April 2009

Version: 2009-4-28

In this chapter, we'll see

- writing our own views,
- using layouts to decorate pages,
- integrating CSS,
- using helpers, and
- linking pages to actions.

## Chapter 7

# Task B: Catalog Display

---

All in all, it's been a successful day so far. We gathered the initial requirements from our customer, documented a basic flow, worked out a first pass at the data we'll need, and put together the maintenance page for the Depot application's products. We even managed to cap off the morning with a decent lunch.

Thus fortified, it's on to our second task. We chatted about priorities with our customer, and she said she'd like to start seeing what the application looks like from the buyer's point of view. Our next task is to create a simple catalog display.

This also makes a lot of sense from our point of view. Once we have the products safely tucked into the database, it should be fairly simple to display them. It also gives us a basis from which to develop the shopping cart portion of the code later.

We should also be able to draw on the work we just did in the product maintenance task—the catalog display is really just a glorified product listing. So, let's get started.

### 7.1 Iteration B1: 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> ruby script/generate controller store index
exists app/controllers/
exists app/helpers/
create app/views/store
exists test/functional/
create app/controllers/store_controller.rb
create test/functional/store_controller_test.rb
create app/helpers/store_helper.rb
create app/views/store/index.html.erb
```

Just 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 (class `StoreController` in the file `store_controller.rb`) containing a single action method, `index`.

So, why did we choose to call our first method `index`? Well, just like most web servers, if you invoke a Rails controller and don't specify an explicit action, Rails automatically invokes the `index` action. In fact, let's try it. Point a browser at <http://localhost:3000/store>, and up pops our web page:<sup>1</sup>



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

Let's start by displaying a simple 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 will 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 will 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 just assume we can ask the model for a list of the products we can sell:

[Download](#) `depot_d/app/controllers/store_controller.rb`

```
class StoreController < ApplicationController
  def index
    @products = Product.find_products_for_sale
  end
end
```

Obviously, this code won't run as it stands. We need to define the method `find_products_for_sale` in the `product.rb` model. The code that follows uses the

1. If you instead see a message saying `No route matches...`, you may need to stop and restart your application at this point. Press `Ctrl-C` in the console window in which you ran `script/server`, and then rerun the command.

Rails `find` method. The `:all` parameter tells Rails that we want all rows that match the given condition. We asked our customer whether she had a preference regarding the order things should be listed, and we jointly decided to see what happened if we displayed the products in alphabetical order, so the code does a sort on title:

def self.xxx  
↔ page 672

[Download](#) depot\_d/app/models/product.rb

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

```
  def self.find_products_for_sale
    find(:all, :order => "title")
  end
```

```
  # validation stuff...
```

The `find` method returns an array containing a `Product` object for each row returned from the database. We use its optional `:order` parameter to have these rows sorted by their title. The `find_products_for_sale` method simply passes this array back to the controller. Note that we made `find_products_for_sale` a class method by putting `self.` in front of its name in the definition. We did this because we want to call it on the class as a whole, not on any particular instance—we'll use it by saying `Product.find_products_for_sale`.

Now we need to write our view template. To do this, edit the file `index.html.erb` 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.)

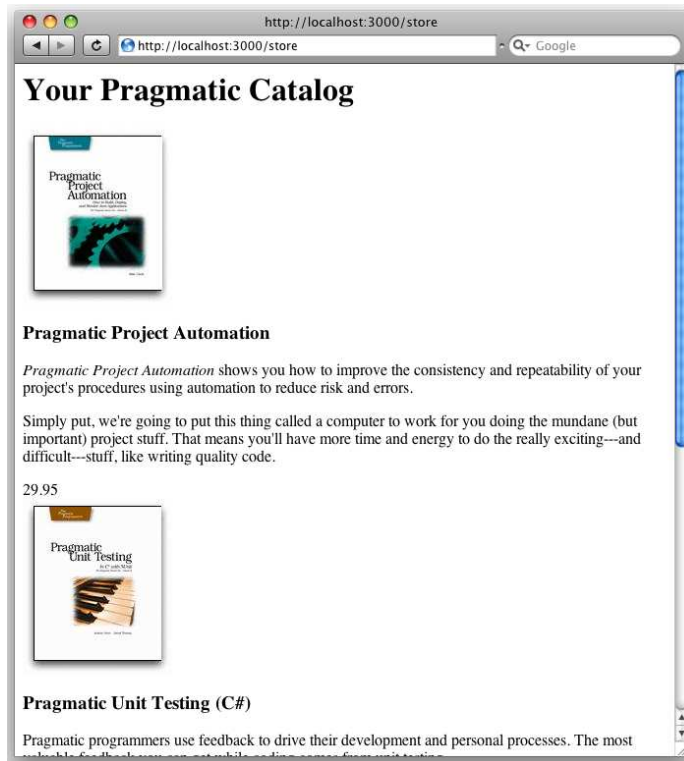
[Download](#) depot\_d/app/views/store/index.html.erb

```
<h1>Your Pragmatic Catalog</h1>
```

```
<% for product in @products -%>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%=h product.title %></h3>
    <%= product.description %>
    <div class="price-line">
      <span class="price"><%= product.price %></span>
    </div>
  </div>
<% end %>
```

This time, we used the `h(string)` method to escape any HTML element in the product title but did not use it to escape the description. This allows us to add HTML stylings to make the descriptions more interesting for our customers.<sup>2</sup>

2. This decision opens a potential security hole, but because product descriptions are created by people who work for our company, we think that the risk is minimal. See Section 27.5, *Protecting Your Application from XSS*, on page 646 for details.




---

Figure 7.1: Our first (ugly) catalog page

---

In general, try to get into the habit of typing `<%=h ... %>` in templates and then removing the `h` only when you've convinced yourself it's safe to do so.

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

Hitting Refresh brings up the display in Figure 7.1. It's pretty ugly, because we haven't yet included the CSS stylesheet. 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 title and sidebar on public-facing pages.

At this point in the real world, we'd probably want to call in the design folks—we've all seen too many programmer-designed websites to feel comfortable inflicting another on the world. 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 an iteration.

## 7.2 Iteration B2: Adding a Page Layout

The pages in a typical website often share a similar layout—the designer will have created a standard template that is used when placing content. Our job is to add this page decoration to each of the store pages.

Fortunately, in Rails we can define layouts. A *layout* is a template into which we can flow additional content. In our case, we can define a single layout for all the store pages and insert the catalog page into that layout. Later we can do the same with the shopping cart and checkout pages. Because there's only one layout, we can change the look and feel of this entire section of our site by editing just one file. This makes us feel better about putting a placeholder in for now; we can update it when the designer eventually returns from the islands.

There are many ways of specifying and using layouts in Rails. We'll choose the simplest for now. If we create a template file in the `app/views/layouts` directory with the same name as a controller, all views rendered by that controller will use that layout by default. So, let's create one now. Our controller is called `store`, so we'll name the layout `store.html.erb`:

[Download](#) depot\_e/app/views/layouts/store.html.erb

```

Line 1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
-       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
-
- <html>
- <head>
5   <title>Pragprog Books Online Store</title>
-   <%= stylesheet_link_tag "depot", :media => "all" %>
- </head>
- <body id="store">
-   <div id="banner">
10    <%= image_tag("logo.png") %>
-    <%= @page_title || "Pragmatic Bookshelf" %>
-   </div>
-   <div id="columns">
-     <div id="side">
15     <a href="http://www...">Home</a><br />
-     <a href="http://www.../faq">Questions</a><br />
-     <a href="http://www.../news">News</a><br />
-     <a href="http://www.../contact">Contact</a><br />
-     </div>
20    <div id="main">
-      <%= yield :layout %>
-    </div>
-   </div>
- </body>
25 </html>

```

Apart from the usual HTML gubbins, this layout has three Rails-specific items. Line 6 uses a Rails helper method to generate a `<link>` tag to our `depot.css`



stylesheet. On line 11, we set the page heading to the value in the instance variable `@page_title`. The real magic, however, takes place on line 21. When we invoke `yield`, passing it the name `:layout`, Rails automatically substitutes in the page-specific content—the stuff generated by the view invoked by this request. In our case, this will be the catalog page generated by `index.html.erb`.<sup>3</sup>

To make this all work, we need to add the following to our `depot.css` stylesheet:

[Download](#) depot\_e/public/stylesheet/depot.css

```
/* Styles for main page */

#banner {
  background: #9c9;
  padding-top: 10px;
  padding-bottom: 10px;
  border-bottom: 2px solid;
  font: small-caps 40px/40px "Times New Roman", serif;
  color: #282;
  text-align: center;
}

#banner img {
  float: left;
}

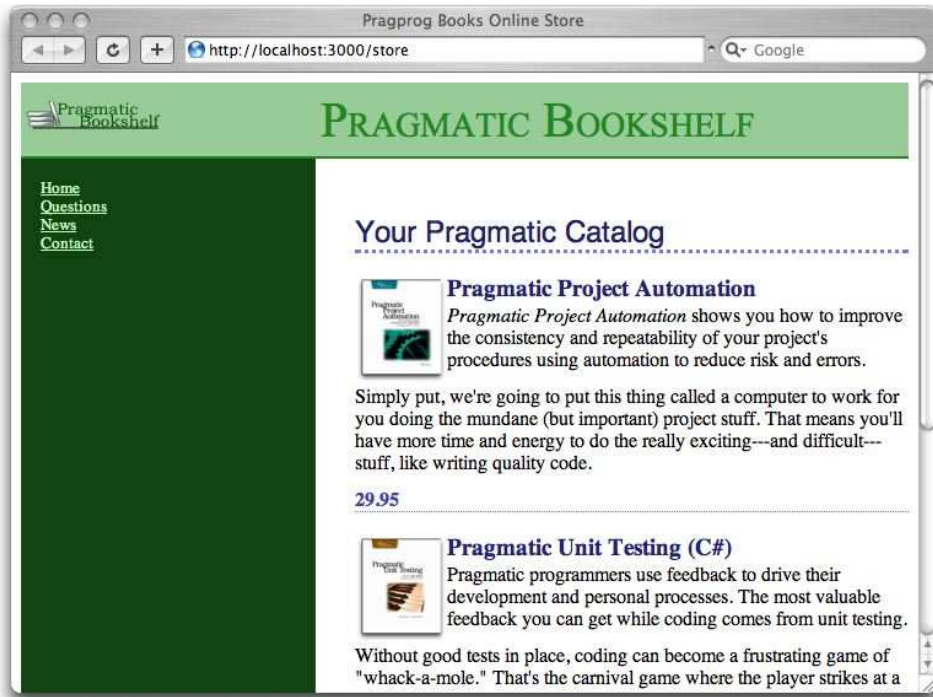
#columns {
  background: #141;
}

#main {
  margin-left: 13em;
  padding-top: 4ex;
  padding-left: 2em;
  background: white;
}

#side {
  float: left;
  padding-top: 1em;
  padding-left: 1em;
  padding-bottom: 1em;
  width: 12em;
  background: #141;
}

#side a {
  color: #bfb;
  font-size: small;
}
```

3. Rails also sets the variable `@content_for_layout` to the results of rendering the action, so you can also substitute this value into the layout in place of the `yield`. This was the original way of doing it (and we personally find it more readable). Using `yield` is considered sexier.




---

Figure 7.2: Catalog with layout added

---

Hit Refresh, and the browser window looks something like Figure 7.2. It won't win any design awards, but it'll show our customer roughly what the final page will look like.

### 7.3 Iteration B3: Using a Helper to Format the Price

There's a problem with our catalog display. The database stores the price as a number, but we'd like to show it as dollars and cents. A price of 12.34 should be shown as \$12.34, and 13 should display as \$13.00.

One solution would be to format the price in the view. For example, we could say this:

```
<span class="price"><%= sprintf("%0.02f", product.price) %></span>
```

This would work, but it embeds knowledge of currency formatting into the view. Should we want to internationalize the application later, this would be a maintenance problem.

Instead, let's use a helper method to format the price as a currency. Rails has an appropriate one built in—it's called `number_to_currency`.

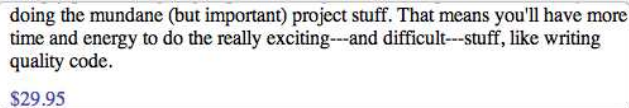
Using our helper in the view is simple: in the index template, we change this:

```
<span class="price"><%= product.price %></span>
```

to the following:

```
<span class="price"><%= number_to_currency(product.price) %></span>
```

Sure enough, when we hit Refresh, we see a nicely formatted price:



doing the mundane (but important) project stuff. That means you'll have more time and energy to do the really exciting---and difficult---stuff, like writing quality code.

\$29.95

## 7.4 Iteration B4: Linking to the Cart

Our customer is really pleased with our progress. We're still on the first day of development, and we have a halfway decent-looking catalog display. However, she points out that we've forgotten a minor detail—there's no way for anyone to buy anything at our store. We forgot to add any kind of *Add to Cart* link to our catalog display.

Back on page 61, we used the `link_to` helper to generate links from a Rails view back to another action in the controller. We could use this same helper to put an *Add to Cart* link next to each product on the catalog page. As we saw on page 95, this is dangerous. The problem is that the `link_to` helper generates an HTML `<a href=...>` tag. When you click the corresponding link, your browser generates an HTTP GET request to the server. And HTTP GET requests are not supposed to change the state of anything on the server—they're to be used only to fetch information.

We previously showed the use of `:method => :delete` as one solution to this problem. Rails provides a useful alternative: the `button_to` method also links a view back to the application, but it does so by generating an HTML form that contains just a single button. When the user clicks the button, an HTTP POST request is generated. And a POST request is just the ticket when we want to do something like add an item to a cart.

Let's add the `Add to Cart` button to our catalog page:

```
Download depot_e/app/views/store/index.html.erb
```

```
<h1>Your Pragmatic Catalog</h1>

<% for product in @products -%>
  <div class="entry">
    <%= image_tag(product.image_url) %>
```

```

    <h3><%=h product.title %></h3>
    <%= product.description %>
    <div class="price-line">
    <span class="price"><%= number_to_currency(product.price) %></span>
    <%= button_to "Add to Cart" %>
    </div>
  </div>
<% 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 a little CSS magic to make them inline:

[Download](#) depot\_f/public/stylesheets/depot.css

```

#store .entry form, #store .entry form div {
  display: inline;
}

```

Now our index page looks like Figure 7.3, on the following page. Of course, if we push the button now, nothing will happen because the button has no action associated with it. So, that's what we will have to fix next.

## What We Just Did

We've put together the basis of the store's catalog display. The steps were as follows:

1. Create a new controller to handle customer-centric interactions.
2. Implement the default index action.
3. Add a class method to the Product model to provide a list of items for sale.
4. Implement a view (an `.html.erb` file) and a layout to contain it (another `.html.erb` file).
5. Use a helper to format prices the way we want.
6. Add a button to each item to allow folks to add it to their carts.
7. Make a simple modification to a stylesheet.

It's time to check it all in and move on to the next task, namely, making the *Add to Cart* link actually do something!

## Playtime

Here's some stuff to try on your own:

- Add a date and time to the sidebar. It doesn't have to update; just show the value at the time the page was displayed.

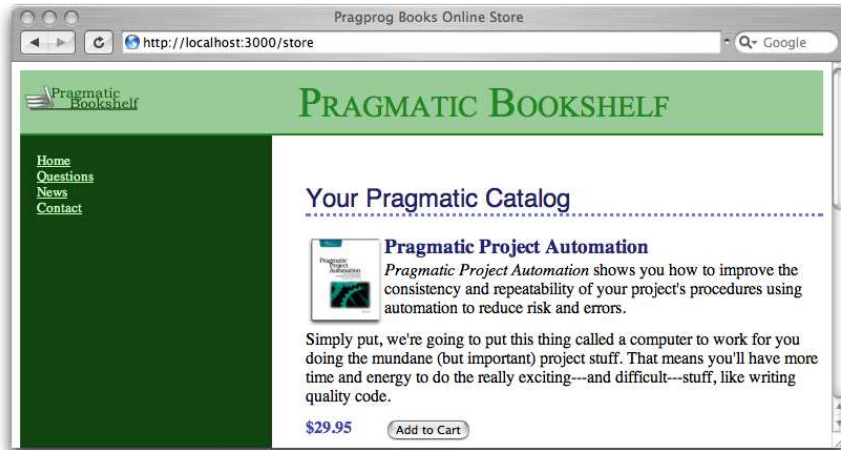


Figure 7.3: Now there's an *Add to Cart* button.

- Change the application so that clicking a book's image will also invoke the yet-to-be-written `add_to_cart` action. Hint: the first parameter to `link_to` is placed in the generated `<a>` tag, and the Rails helper `image_tag` constructs an HTML `<img>` tag. Include a call to it as the first parameter to a `link_to` call. Be sure to include `:method => :post` in your `html_options` on your call to `link_to`.
- The full description of the `number_to_currency` helper method is as follows:  
`number_to_currency(number, options = {})`

*Formats a number into a currency string. The options hash can be used to customize the format of the output. The number can contain a level of precision using the `:precision` key; the default is 2. The currency type can be set using the `:unit` key (default "\$") The unit separator can be set using the `:separator` key (default ".") The delimiter can be set using the `:delimiter` key (default ",").*

```
number_to_currency(1234567890.50)    -> $1,234,567,890.50
number_to_currency(1234567890.506)  -> $1,234,567,890.51
number_to_currency(1234567890.50, :unit => "&pound;",
                                   :separator => ",", :delimiter => "'")
                                   -> &pound;1234567890,50
```

Experiment with setting various options, and see the effect on your catalog listing.

(You'll find hints at <http://pragprog.wikidot.com/rails-play-time>.)

# The Pragmatic Bookshelf

---

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

## Visit Us Online

---

### Agile Web Development with Rails

<http://pragprog.com/titles/rails3>

Source code from this book, errata, and other resources. Come give us feedback, too!

### Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

### Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

### New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments in the news.

## Buy the Book

---

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: [pragmaticprogrammer.com/titles/rails3](http://pragmaticprogrammer.com/titles/rails3).

## Contact Us

---

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	<a href="http://www.pragmaticprogrammer.com/catalog">www.pragmaticprogrammer.com/catalog</a>
Customer Service:	<a href="mailto:orders@pragmaticprogrammer.com">orders@pragmaticprogrammer.com</a>
Non-English Versions:	<a href="mailto:translations@pragmaticprogrammer.com">translations@pragmaticprogrammer.com</a>
Pragmatic Teaching:	<a href="mailto:academic@pragmaticprogrammer.com">academic@pragmaticprogrammer.com</a>
Author Proposals:	<a href="mailto:proposals@pragmaticprogrammer.com">proposals@pragmaticprogrammer.com</a>