Extracted from:

Agile Web Development with Rails 5

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rails 5

Agile Web Development with Rails 5

Sam Ruby
Dave Thomas
David Heinemeier Hansson

Edited by Susannah Davidson Pfalzer

RAILS

Agile Web Development with Rails 5

Sam Ruby
Dave Thomas
David Heinemeier Hansson

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (index)
Eileen Cohen (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-171-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2016

In this chapter, you'll see:

- Adding new classes to your application
- Adding a new templating language

CHAPTER 25

Rails Plugins

Since the beginning of this book, we've talked incessantly about convention over configuration in that Rails has sensible defaults for just about everything. And more recently in the book, we've described Rails in terms of the underlying gems that you get when you install Rails. Now it is time to put those two thoughts together and reveal that the initial set of gems that Rails provides you with is a sensible set of defaults—a group of gems that you can both add to and change.

With Rails, gems are the primary way in which you *plug in* new functionality. Instead of describing this in the abstract, we will select a few plugins and use them to illustrate different aspects of how plugins are installed and what plugins can do. The fact that many of these plugins turn out to be immediately useful for your day-to-day work is simply a bonus!

Let's start with a simple plugin that can make you money.

Credit Card Processing with Active Merchant

In Iteration G1 on page ? we mentioned that we were temporarily punting on handling credit cards. Being able to charge a customer is clearly an important part of taking an order. Although this functionality isn't built into the core of Rails, there is a gem that provides this.

You've already seen how you control what gems get loaded by your application; you do this by editing your Gemfile. Since we are going to cover a number of such gems in this chapter, let's add all of the ones that we'll cover at once. Add these any place you like; we've chosen to do so at the end of the file:

rails50/depot_w/Gemfile

```
gem 'activemerchant', '~> 1.58'
gem 'haml', '~> 4.0'
gem 'kaminari', '~> 0.16'
```

You will note that we follow best practices by specifying a minimum version and effectively specifying an upper bound on the version number so that this demo will pick a version that is unlikely to contain an incompatible change.

As for the gems we added, we will cover each in a separate section. This section will focus on Active Merchant.[1]

With this in place, we can use the bundle command to install our dependencies:

```
depot> bundle install
```

Depending on your operating system and your setup, you may need to run this command as root.

The bundle command will actually do much more. It will cross-check gem dependencies, find a configuration that works, and download and install whatever components are necessary. But this needn't concern us now; we added only one component, and we can rest assured that this one is included in the gems that the bundler installed.

We must do one last thing after updating or installing a new gem: restart the server. Although Rails does a good job of detecting and keeping up with your latest changes to your application, it is impossible to predict what needs to be done when an entire gem is added or replaced. We won't be using the server in this section but will shortly. Make sure that the server is running the Depot application.

To demonstrate this functionality, we will create a small script, which we will place in the script directory:

rails50/depot_w/script/creditcard.rb

```
credit_card = ActiveMerchant::Billing::CreditCard.new(
  number:     '4111111111111111',
  month:      '8',
  year:       '2009',
  first_name: 'Tobias',
  last_name:  'Luetke',
  verification_value:  '123'
)

puts "Is #{credit_card.number} valid?  #{credit_card.valid?}"
```

There is not much to this script. It creates an instance of an ActiveMerchant::Billing::CreditCard class and then calls valid?() on this object. Let's run it:

```
$ rails runner script/creditcard.rb
Is 4111111111111111 valid?  false
```

1. http://www.activemerchant.org/

There's not much to it; it just worked. Note that no require statements were necessary; simply listing the gem you want in your Gemfile makes the function available to your application.

At this point, you should be able to see how you could use this functionality in the Depot application. You know how to add a field to the Orders table via a migration. You know how to add that field to the view. You know how to add validation logic to your model, which calls the valid?() method that we used earlier. If you go to the merchant site, you can even find out how to authorize() and capture() a payment, though this does require you to have a login and a password with an existing commerce gateway. Once that is set up, you know how to call this logic from your controller.

Just think: that was made possible by adding a single line to your Gemfile.

As we stated at the beginning of this chapter, adding gems to your Gemfile is the preferred way to extend Rails. The advantages of doing so are numerous: all of your dependencies are tracked by Bundler, are all preloaded for immediate use by your application, and can be packed for easy deployment.

This was a simple addition. Let's try something more significant, something that provides an alternative to one of the gems that Rails depends on.

Beautifying Our Markup with Haml

Let's take a look once again at a simple view that we use in the Depot application, in this case, one that presents our storefront:

rails50/depot_v/app/views/store/index.html.erb

```
<p id="notice"><%= notice %></p>
<h1><%= t('.title_html') %></h1>
<% cache @products do %>
  <% @products.each do |product| %>
    <% cache 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 t('.add_html'),
            line_items_path(product_id: product, locale: I18n.locale),
            remote: true %>
        </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

This code gets the job done. It contains the basic HTML, with interspersed bits of Ruby code enclosed in <% and %> markup. Inside that markup, an equal sign is used to indicate that the value of the expression is to be converted to HTML and displayed.

This is not only an adequate solution to the problem at hand; it is also all that is really needed for a large number of Rails applications. Additionally, it is an ideal place to start for books—like this one—where some knowledge of HTML may be presumed, but many of the readers are new to Rails and often to Ruby. The last thing you would want to do in that situation is to introduce yet another new language.

But now that you are past that learning curve, let's explore a new language —one that more closely integrates the production of markup with Ruby code, namely, HTML Abstraction Markup Language (Haml).

To start with, let's remove the file we just looked at:

```
$ rm app/views/store/index.html.erb
```

In its place, let's create a new file:

rails50/depot_w/app/views/store/index.html.haml

```
%p#notice= notice

%h1= t('.title_html')

- cache @products do
  - @products.each do |product|
    - cache product do
      .entry
        = image_tag(product.image_url)
        %h3= product.title
        = sanitize(product.description)
        .price_line
          %span.price= number_to_currency(product.price)
          = button_to t('.add_html'),
            line_items_path(product_id: product, locale: I18n.locale),
            remote: true
```

Note the new extension: .html.haml. This indicates that the template is a Haml template instead of an ERB template.

The first thing you should notice is that the file is considerably smaller. Here's a quick overview of what is going on, based on what the first character is on each line:

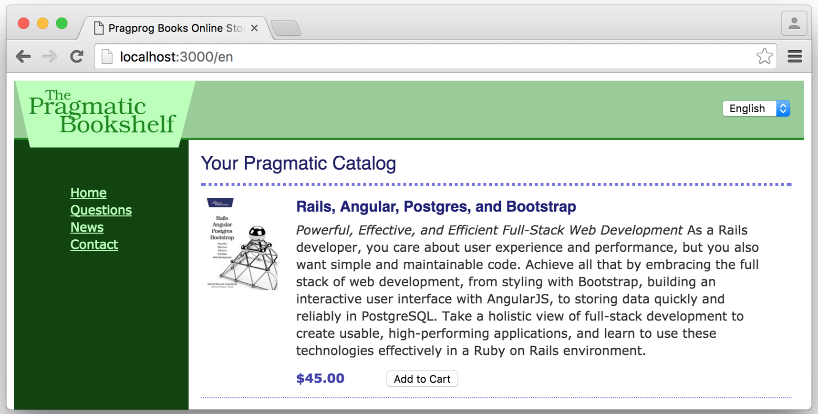
- Dashes indicate a Ruby statement that does not produce any output
- Percent signs (%) indicate an HTML element.

- Equal signs (=) indicate a Ruby expression that does produce output to be displayed. This can be used either on lines by themselves or following HTML elements.
- Dots (.) and hash (#) characters may be used to define class and id attributes, respectively. This can be combined with percent signs or used stand-alone. When used by itself, a div element is implied.
- A comma at the end of a line containing an expression implies a continuation. In the prior example, the button_to() call is continued across two lines.

An important thing to note is that indentation is important in Haml. Returning to the same level of indentation closes the if statement, loop, or tag that is currently open. In this example, the paragraph is closed before the h1, the h1 is closed before the first div, but the div elements nest, with the first containing an h3 element and the second containing both a span and a button_to().

As you can also see, all of your familiar helpers are available, things like t(), image_tag(), and button_to(). In every meaningful way, Haml is as integrated into your application as ERB is. You can mix and match: you can have some templates using ERB and others using Haml.

As you have already installed the Haml gem, there truly is nothing more you need to do. To see this in action, all you need to do is to visit your storefront. What you should see should match the following screenshot.



Your Pragmatic Catalog

Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development As a Rails developer, you care about user experience and performance, but you also want simple and maintainable code. Achieve all that by embracing the full stack of web development, from styling with Bootstrap, building an interactive user interface with AngularJS, to storing data quickly and reliably in PostgreSQL. Take a holistic view of full-stack development to create usable, high-performing applications, and learn to use these technologies effectively in a Ruby on Rails environment.

$45.00 Add to Cart

If that looks unremarkable, that’s because it should look *exactly* like it did before. And that, if you think about it, is all the more remarkable because the application layout continues to be implemented as an ERB template and the index is implemented using Haml. Despite this, everything integrates seamlessly and effortlessly.

Although this clearly is a deeper level of integration than simply adding a task or a helper, it still is an addition. Next, let’s explore a plugin that changes a core object in Rails.