Extracted from:

Agile Web Development with Rails

Second Edition

In this chapter, we'll see

- using "has_many :through" join tables
- creating a REST interface
- generating XML using rxml templates
- generating XML using to_xml on model objects
- handling requests for different content types
- creating application documentation
- getting statistics on our application

Chapter 12

Task G: One Last Wafer-Thin Change

Over the days that followed our first few iterations, we added fulfillment functionality to the shopping system and rolled it out. It was a great success, and over the months that followed the Depot application became a core part of the business. So much so, in fact, that the marketing people got interested. They want to send mass mailings to people who have bought particular books, telling them that new titles are available. They already have the spam^H^H^H^Hmailing system; it just needs an XML feed containing customer names and e-mail addresses.

12.1 Generating the XML Feed

Let's set up a REST-style interface to our application. REST stands for REpresentational State Transfer, which is basically meaningless. What it really means is that you use HTTP verbs (GET, POST, DELETE, and so on) to send requests and responses between applications. In our case, we'll let the marketing system send us an HTTP GET request, asking for the details of customers who've bought a particular product. Our application will respond with an XML document.[1] We talk with the IT folks over in marketing, and they agree to a simple request URL format.

http://my.store.com/info/who_bought/*<product id>*

So, we have two issues to address: we need to be able to find the customers who bought a particular product, and we need to generate an XML feed from that list. Let's start by generating the list.

Navigating Through Tables

Figure 12.1, on the next page, shows how the orders side of our database is currently structured. Every order has a number of line items, and each line

1. We *could* have used web services to implement this transfer—Rails has support for acting as both a SOAP and XML-RPC client and server. However, this seems like overkill in this case.

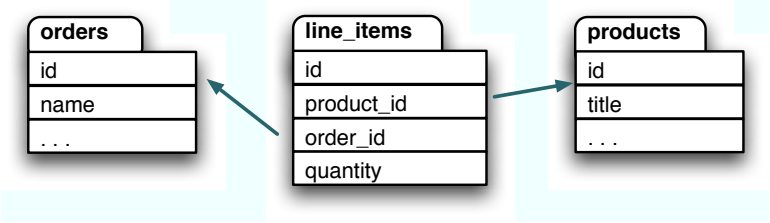

Figure 12.1: Database Structure

item is associated with a product. Our marketing folks want to navigate these associations in the opposite direction, going from a particular product to all the line items that reference that product and then from these line items to the corresponding order.

As of Rails 1.1, we can do this using a :through relationship. We can add the following declaration to the product model.

Download depot_q/app/models/product.rb

```
class Product < ActiveRecord::Base

  has_many :orders, :through => :line_items

  # . . .
```

Previously we used has_many to set up a parent/child relationship between products and line items: we said that a product has many line items. Now, we're saying that a product is also associated with many orders but that there's no direct relationship between the two tables. Instead, Rails knows that to get the orders for a product, it must first find the line items for the product and then find the order associated with each line item.

Now this might sound fairly inefficient. And it would be, if Rails first fetched the line items and then looped over each to load the orders. Fortunately, it's smarter than that. As you'll see if you look at the log files when we run the code we're about to write, Rails generates an efficient SQL join between the tables, allowing the database engine to optimize the query.

With the :through declaration in place, we can find the orders for a particular product by referencing the orders attribute of that product.

```
product = Product.find(some_id)
orders = product.orders
logger.info("Product #{some_id} has #{orders.count} orders")
```

Creating a REST Interface

Anticipating that this won't be the last request that the marketing folks make, we create a new controller to handle informational requests.

```
depot> ruby script/generate controller info
exists  app/controllers/
exists  app/helpers/
create  app/views/info
exists  test/functional/
create  app/controllers/info_controller.rb
create  test/functional/info_controller_test.rb
create  app/helpers/info_helper.rb
```

We'll add the who_bought action to the info controller. It simply loads up the list of orders given a product id.

```
def who_bought
  @product = Product.find(params[:id])
  @orders  = @product.orders
end
```

Now we need to implement the template that returns XML to our caller. We could do this using the same rhtml templates we've been using to render web pages, but there are a couple of better ways. The first uses rxml templates, designed to make it easy to create XML documents. Let's look at the template who_bought.rxml, which we create in the app/views/info directory.

Download depot_q/app/views/info/who_bought.rxml

```
xml.order_list(:for_product => @product.title) do
  for o in @orders
    xml.order do
      xml.name(o.name)
      xml.email(o.email)
    end
  end
end
```

Believe it or not, this is just Ruby code. It uses Jim Weirich's Builder library, which generates a well-formed XML document as a side effect of executing a program.

Within an rxml template, the variable xml represents the XML object being constructed. When you invoke a method on this object (such as the call to order_list on the first line in our template), the builder emits the corresponding XML tag. If a hash is passed to one of these methods, it's used to construct the attributes to the XML tag. If you pass a string, it is used as the tag's value.

If you want to nest tags, pass a block to the outer builder method call. XML elements created inside the block will be nested inside the outer element. We use this in our example to embed a list of <*order*> tags inside an <*order_list*> and then to embed a <*name*> and <*email*> tag inside each <*order*>.

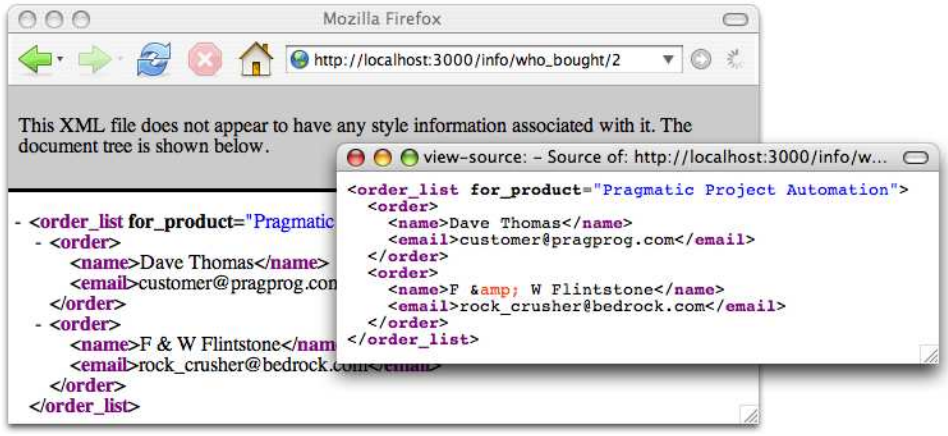

Figure 12.2: XML Returned by the who_bought Action

We can test this method using a browser or from the command line. If you enter the URL into a browser, the XML will be returned. How it is displayed depends on the browser: on my Mac, Safari renders the text and ignores the tags, while Firefox shows a nicely highlighted representation of the XML (as shown in Figure 12.2). In all browsers, the View → Source option should show exactly what was sent from our application.

You can also query your application from the command line using a tool such as curl or wget.

```
depot> curl http://localhost:3000/info/who_bought/1
<order_list for_product="Pragmatic Project Automation">
  <order>
    <name>Dave Thomas</name>
    <email>customer@pragprog.com</email>
  </order>
  <order>
    <name>F &amp;amp; W Flintstone</name>
    <email>rock_crusher@bedrock.com</email>
  </order>
</order_list>
```

In fact, this leads to an interesting question: can we arrange our action so that a user accessing it from a browser sees a nicely formatted list, while those making a REST request get XML back?

Responding Appropriately

Requests come into a Rails application using HTTP. An HTTP message consists of some headers and (optionally) some data (such as the POST data from a

form). One such header is Accept:, which the client uses to tell the server the types of content that may be returned. For example, a browser might send an HTTP request containing the header

```
Accept:  text/html, text/plain, application/xml
```

In theory, a server should respond only with content that matches one of these three types.

We can use this to write actions that respond with appropriate content. For example, we could write a who_bought action that uses the accept header. If the client accepts only XML, then we could return an XML-format REST response. If the client accepts HTML, then we can render an HTML page instead.

In Rails, we use the respond_to method to perform conditional processing based on the Accepts header. First, let's write a trivial template for the HTML view.

Download depot_r/app/views/info/who_bought.rhtml

```
<h3>People Who Bought <%= @product.title %></h3>

<ul>
  <% for order in @orders  -%>
    <li>
        <%= mail_to order.email, order.name %>
    </li>
  <%  end -%>
</ul>
```

Now we'll use respond_to to vector to the correct template depending on the incoming request accept header.

Download depot_r/app/controllers/info_controller.rb

```
def who_bought
  @product = Product.find(params[:id])
  @orders  = @product.orders
  respond_to do |format|
    format.html
    format.xml
  end
end
```

Inside the respond_to block, we list the content types we accept. You can think of it being a bit like a case statement, but it has one big difference: it ignores the order you list the options in and instead uses the order from the incoming request (because the client gets to say which format it prefers).

Here we're using the default action for each type of content. For html, that action is to invoke render. For xml, the action is to render the .rxml template. The net effect is that the client can select to receive either HTML or XML from the same action.

Unfortunately, this is hard to try with a browser. Instead, let's use a command-line client. Here we use curl (but tools such as wget work equally as well). The -H option to curl lets us specify a request header. Let's ask for XML first.

```
depot> curl -H "Accept: application/xml" \
            http://localhost:3000/info/who_bought/1
<order_list for_product="Pragmatic Project Automation">
  <order>
    <name>Dave Thomas</name>
    <email>customer@pragprog.com</email>
  </order>
  <order>
    <name>F &amp;amp; W Flintstone</name>
    <email>crusher@bedrock.com</email>
  </order>
</order_list>
```

And then HTML.

```
depot> curl -H "Accept: text/html" \
         http://localhost:3000/info/who_bought/1
<h3>People Who Bought Pragmatic Project Automation</h3>
<ul>
  <li>
    <a href="mailto:customer@pragprog.com">Dave Thomas</a>
  </li>
  <li>
    <a href="mailto:crusher@bedrock.com">F &amp; W Flintstone</a>
  </li>
</ul>
```

Another Way of Requesting XML

Although using the Accept header is the "official" HTTP way of specifying the content type you'd like to receive, it isn't always possible to set this header from your client. Rails provides an alternative: we can set the preferred format as part of the URL. If we want the response to our who_bought request to come back as HTML, we can ask for /info/who_bought/1.html. If instead we want XML, we can use /info/who_bought/1.xml. And this is extensible to any content type (as long as we write the appropriate handler in our respond_to block).

Try requesting the URL http://localhost:3000/info/who_bought/1.xml. Depending on your browser, you might see a nicely formatted XML display, or you might see a blank page. If you see the latter, use your browser's View → Source function to have a look at the response.

Autogenerating the XML

In the previous examples, we generated the XML responses by hand, using the rxml template. That gives us control over the order of the elements returned. But if that order isn't important, we can let Rails generate the XML for a model

object for us by calling the model's to_xml method. In the code that follows, we've overridden the default behavior for XML requests to use this.

```
def who_bought
  @product = Product.find(params[:id])
  @orders  = @product.orders
  respond_to do |accepts|
    accepts.html
    accepts.xml { render :xml => @product.to_xml(:include => :orders) }
  end
end
```

The :xml option to render tells it to set the response content type to application/xml. The result of the to_xml call is then sent back to the client. In this case, we dump out the @product variable and any orders that reference that product.

```
depot> curl http://localhost:3000/info/who_bought/1.xml
<?xml version="1.0" encoding="UTF-8"?>
<product>
  <image-url>/images/auto.jpg</image-url>
  <title>Pragmatic Project Automation</title>
  <price type="integer">2995</price>
  <orders>
    <order>
      <name>Dave Thomas</name>
      <id type="integer">1</id>
      <pay-type>check</pay-type>
      <address>123 The Street</address>
      <email>customer@pragprog.com</email>
    </order>
    <order>
      <name>F &amp; W Flintstone</name>
      <id type="integer">2</id>
      <pay-type>check</pay-type>
      <address>123 Bedrock</address>
      <email>crusher@bedrock.com</email>
    </order>
  </orders>
  <id type="integer">1</id>
  <description>&lt;p&gt;
    &lt;em&gt;Pragmatic Project Automation&lt;/em&gt; shows
    you how to improve the consistency and repeatability of
    your project's procedures using automation to reduce risk
    and errors.  &lt;/p&gt; &lt;p&gt; Simply put, we're going
    to put this thing called
    a computer to work for you 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.
    &lt;/p&gt;
  </description>
</product>
```

Note that by default to_xml dumps everything out. You can tell it to exclude certain attributes, but that can quickly get messy. If you have to generate XML that meets a particular schema or DTD, you're probably better off sticking with rxml templates.

12.2 Finishing Up

The coding is over, but we can still do a little more tidying before we deploy the application into production.

We might want to check out our application's documentation. As we've been coding, we've been writing brief but elegant comments for all our classes and methods. (We haven't shown them in the code extracts in this book because we wanted to save space.) Rails makes it easy to run Ruby's RDoc utility on all the source files in an application to create good-looking programmer documentation. But before we generate that documentation, we should probably create a nice introductory page so that future generations of developers will know what our application does. To do this, edit the file doc/README_FOR_APP, and enter anything you think might be useful. This file will be processed using RDoc, so you have a fair amount of formatting flexibility.

RDoc
↪ page 645

You can generate the documentation in HTML format using the rake command.

```
depot> rake doc:app
```

This generates documentation into the directory doc/app. Figure 12.3, on the next page, shows the initial page of the output generated.

Finally, we might be interested to see how much code we've written. There's a Rake task for that, too. (Your numbers will be different from this, if for no other reason than you probably won't have written tests yet. That's the subject of the next chapter.)

```
depot> rake stats
(in /Users/dave/Work/depot)
+----------------------+-------+-------+---------+---------+-----+-------+
| Name                 | Lines |   LOC | Classes | Methods | M/C | LOC/M |
+----------------------+-------+-------+---------+---------+-----+-------+
| Helpers              |    17 |    15 |       0 |       1 |   0 |    13 |
| Controllers          |   229 |   154 |       5 |      23 |   4 |     4 |
| Components           |     0 |     0 |       0 |       0 |   0 |     0 |
|   Functional tests   |   206 |   141 |       8 |      25 |   3 |     3 |
| Models               |   261 |   130 |       6 |      18 |   3 |     5 |
|   Unit tests         |   178 |   120 |       5 |      13 |   2 |     7 |
| Libraries            |     0 |     0 |       0 |       0 |   0 |     0 |
|   Integration tests  |   192 |   130 |       2 |      10 |   5 |    11 |
+----------------------+-------+-------+---------+---------+-----+-------+
| Total                |  1083 |   690 |      26 |      90 |   3 |     5 |
+----------------------+-------+-------+---------+---------+-----+-------+
  Code LOC: 299     Test LOC: 391     Code to Test Ratio: 1:1.3
```

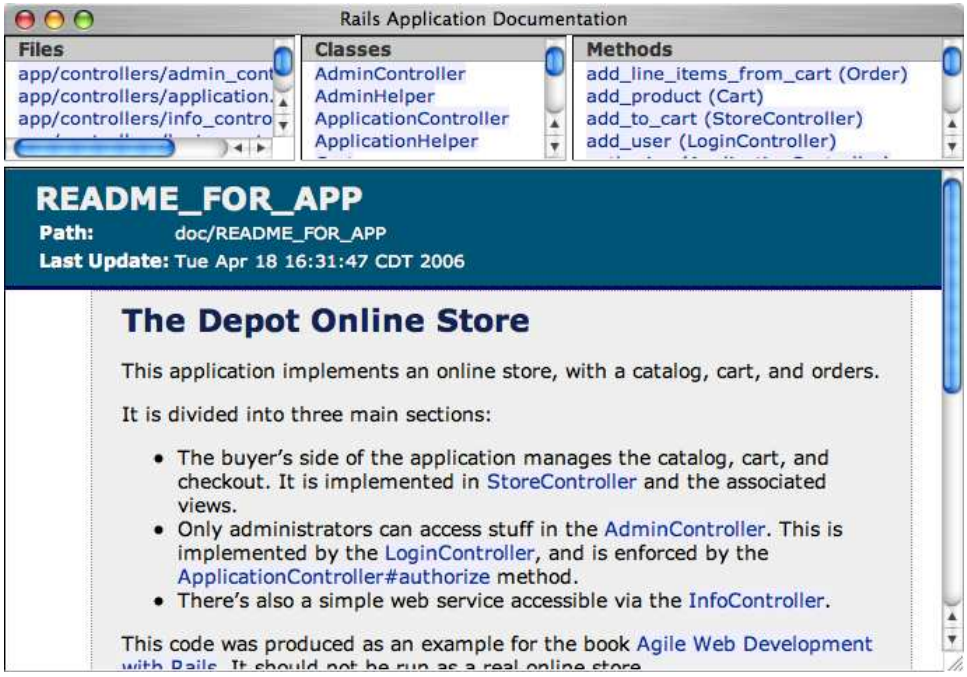

Figure 12.3: Our Application's Internal Documentation

Playtime

Here's some stuff to try on your own.

- Change the original catalog display (the index action in the store controller) so that it returns an XML product catalog if the client requests an XML response.
- Try using rxml templates to generate normal HTML (technically, XHTML) responses. What are the advantages and disadvantages?
- If you like the programmatic generation of HTML responses, have a look at Markaby.[2] It installs as a plugin, so you'll be trying stuff we haven't talked about yet, but the instructions on the web site are clear.
- Add credit card and PayPal processing, fulfillment, couponing, RSS support, user accounts, content management, and so on, to the Depot application. Sell the resulting application to a big-name web company. Retire early, and do good deeds.

(You'll find hints at http://wiki.pragprog.com/cgi-bin/wiki.cgi/RailsPlayTime)

2. http://redhanded.hobix.com/inspect/markabyForRails.html

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://pragmaticprogrammer.com/titles/rails2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragmaticprogrammer.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragmaticprogrammer.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://pragmaticprogrammer.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/rails2.

Contact Us

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