CHAPTER 4

Book Catalog Browsing

In this chapter, we'll work through setting up the basic functionality of a book catalog from the customer's perspective. We'll build the chapter around four user stories where Jill, George's book-hogging customer, plays the starring role.

For the Emporium book catalog, we will create a simple catalog page for the books, along with pages that display details for individual titles. The interface also will need a way to search for books by their titles and descriptions. We will use Ferret, a full-text search engine written in Ruby, to supply this functionality. Additionally, we will create a latest books page and RSS feed, so that Jill can follow what's new at Emporium.

Getting the Book Catalog Requirements

If there's one person keeping Emporium going, that's Jill. Jill lives just a couple of blocks away from the store. When she rushes through the door with her plasma-TV-sized goggles, George knows that the day is saved.

However, Jill's health is not as it used to be. Her visits have gotten fewer and fewer lately. She would love to support George and buy a lot of new books, but it's just too much effort for her to come over daily. Jill is a smart lady, though, and she's found out that this new thing called the Internet can work as an intermediary between her and her beloved book supply.

To make Jill a happy online customer, George comes up with four user stories for this sprint:

- *Browse books*: Jill needs a way to browse the books in the shop. We will keep the list really simple at this point, just letting her shuffle through the supply and find out about new titles.
- *View book details*: After browsing through titles in the first story or getting a list of matching titles in the second one, Jill needs a way to get specific information about a particular title. As a former librarian, she is obsessed about knowing even the most mundane details of every book she is thinking about buying.
- *Search books*: Sometimes Jill finds out about an interesting topic and wants to know more about the subject. She needs to be able to write a few keywords and get a list of all the titles that match her search.
- *Get latest books*: As a book addict, Jill needs a way to keep current about all new books. She would like to find out about new titles with a single look on the Emporium site. What would make her really happy, however, would be an RSS feed that she could follow on her shiny white iBook without even visiting the website. That would leave her more time for her real pleasure, perusing her precious tomes.

We will tackle these user stories in this chapter, one by one, using the already familiar TDD method.

Implementing the Book Catalog Interface

To be able to really test browsing a list of titles, we need to have a number of books available for viewing. Therefore, we need to expand our `authors.yml`, `publishers.yml`, `books.yml`, and `authors_books.yml` fixture files in `test/fixtures`. You can download the files from the Source Code/Downloads section of `www.apress.com`.

As in the previous chapter, we'll use integration tests for this sprint, because they work well to exercise the book catalog browsing system from end to end. First, we'll create a test stub by using the Rails test generator:

```
$ script/generate integration_test BrowsingAndSearching
```

```
      exists  test/integration/
      create  test/integration/browsing_and_searching_test.rb
```

Again, we'll delete the `test_truth` from the test file and replace it with our real test, as shown in Listing 4-1.

Listing 4-1. *First Version of the Integration Test for the Book Catalog Interface*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BrowsingAndSearchingTest < ActionController::IntegrationTest
  fixtures :publishers, :authors, :books, :authors_books

  def test_browsing_the_site
    jill = enter_site(:jill)
    jill.browse_index
  end

  private

  module BrowsingTestDSL
    attr_writer :name

    def browse_index
      get "/catalog"

      assert_response :success
      assert_template "catalog/index"
      assert_tag      :tag => "dl", :attributes =>
                        { :id => "books" },
                      :children =>
                        { :count => 10, :only =>
                          {:tag => "dt"}}
      assert_tag :tag => "dt", :content => "The Idiot"
    end
  end

  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

In the test case, we create a DSL module, as in the previous chapter. We first test that requesting "/catalog" works and that we get the default index template rendered. Then we check that there is a list of books (laid out with an HTML definition list) on the page and that a book called *The Idiot* exists in that list.

Our test fails magnificently, in plain old TDD way, so it's time to implement the functionality.

First, we need to create the controller for the catalog. We'll name it catalog so that it will match the URL requested in the browse_site test method. Once again, use the familiar script/generate command and give the needed action names (for all our projected user stories) as parameters.

```
$ script/generate controller Catalog index show search latest
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/catalog
      exists  test/functional/
      create  app/controllers/catalog_controller.rb
      create  test/functional/catalog_controller_test.rb
      create  app/helpers/catalog_helper.rb
      create  app/views/catalog/index.rhtml
      create  app/views/catalog/show.rhtml
      create  app/views/catalog/search.rhtml
      create  app/views/catalog/latest.rhtml
```

Implementing the Browse Books User Story

Now that we have the controller in place, we're ready to begin with the Browse Books user story.

Modifying the Controller

At this point, we're interested in only the index action and the corresponding view. Open app/controllers/catalog_controller.rb, which was just created by the generator command. Modify the index method so that it looks as follows:

```
def index
  @page_title = "Book List"
  @book_pages, @books = paginate :books,
                          :per_page => 10,
                          :include => [:authors, :publisher],
                          :order => "books.id desc"
end
```

In the `index` action, we first set the page title so that the layout file will pick it up and show it in the headers of the resulting page. Additionally, the action contains a normal pagination call, just as in Chapter 3. However, this time, we use the `include` parameter for the `paginate` call.

The `include` parameter is used in the ActiveRecord `find` method (which is used internally by `paginate`) to make ActiveRecord build up a join query. This single SQL query will be used not only to find the books, but also to fetch the associated authors and publishers from the database. If we omitted the parameter, our code would end up calling a new SQL query each time we needed to get the author or publisher details for a given book. In our case, it would result in $2n+1$ (where n is the number of books) queries instead of just one. When the site gets more traffic, that could become a huge performance bottleneck.

■ **Note** We can hear you ask, "Where does the $2n+1$ come from?" The first query is the one where all the books are fetched. Then, when we iterate over all the n books and call their `authors` and `publisher` methods, each call will result in an additional SQL query, resulting in two additional queries for each book. The resulting amount of queries is thus 2 queries $\times$ n books + the original query, or $2n+1$.

Modifying the View

Next, open `app/views/catalog/index.rhtml` and replace its contents with the following code.

```
<dl id="books">
  <% for book in @books %>
    <dt><%= book.title %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>


<%= link_to 'Previous page', { :page => @book_pages.current.previous } if ➥
@book_pages.current.previous %>
<%= link_to 'Next page', { :page => @book_pages.current.next } if ➥
@book_pages.current.next %>
```

In the view, we iterate over all the books we got from the controller and show their titles, authors, prices, page counts, and publishers. The `pluralize` helper will show the word "page" in either singular or plural, depending on the value of `book.page_count`. In the end, we show links to next and/or previous page in case there are more than ten books in the `@books` array.

Running the Integration Test

Now that we have our simple browsing functionality implemented, we can run our test case.

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
.
Finished in 0.514885 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

The test passes, but browsing is really not browsing if it involves only a single page. So, let's create another test case that checks that the pagination in our catalog works as expected. Make the following changes to `test/integration/browsing_and_searching_test.rb`:

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BrowsingAndSearchingTest < ActionController::IntegrationTest
  fixtures :publishers, :authors, :books, :authors_books

  def test_browsing_the_site
    jill = enter_site(:jill)
    jill.browse_index
    jill.go_to_second_page
  end

  private

  module BrowsingTestDSL
    attr_writer :name

    def browse_index
      get "/catalog"

      assert_response :success
      assert_template "catalog/index"
      assert_tag      :tag => "dl", :attributes =>
                        { :id => "books" },
                      :children =>
                        { :count => 10, :only =>
                          {:tag => "dt"}}
      assert_tag :tag => "dt", :content => "The Idiot"
    end
```

```
    def go_to_second_page
      get "/catalog?page=2"
      assert_response :success
      assert_template "catalog/index"
      assert_equal    Book.find_by_title("Pro Rails E-Commerce"),
                      assigns(:books).last
    end
  end

  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

In `go_to_second_page`, we first fetch the second catalog page. We then check that we get a normal response and the correct template in return. Finally, we check that the first one of the books in our `books.yml` fixture file is on this page, since the books are ordered in a descending chronological order on the catalog page. Running the tests again confirms that the catalog page is working as expected:

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
.
Finished in 0.110837 seconds.

1 tests, 7 assertions, 0 failures, 0 errors
```

Now that we have a working catalog page, it would be nice to make it the home page of the whole book store. We already briefly mentioned Rails routes in Chapter 2, and now we're going to take advantage of them again. Open `config/routes.rb` and change the line for default root url to look like this:

```
# You can have the root of your site routed by hooking up "
# -- just remember to delete public/index.html.
map.connect ", :controller => "catalog"
```

This means that all the requests for the root url are routed to the default action (`index`) of `CatalogController`.

Implementing the View Book Details User Story

Having a catalog page for a series of books is nice, but it's not suitable for excruciating details about every item. Therefore, the next thing for us to do is to implement a page for individual titles. As always, we start by writing a test for this story.

We already have a test case, so we can just extend that. In `test/integration/browsing_and_searching_test.rb`, we'll add another chapter to the story of Jill, right below `test_browsing_the_site`:

```
def test_getting_details
  jill = enter_site(:jill)
  jill.get_book_details_for "Pride and Prejudice"
end
```

Then we add a new method to our `BrowsingTestDSL` module to keep the test code clean:

```
def get_book_details_for(title)
  @book = Book.find_by_title(title)
  get "/catalog/show/#{@book.id}"
  assert_response :success
  assert_template "catalog/show"

  assert_tag      :tag => "h1",
                  :content => @book.title
  assert_tag      :tag => "h2",
                  :content => "by #{@book.authors.map{|a| a.name}}"
end
```

The `get_book_details_for` method simply fetches a book with the given name from the database, then requests the corresponding `show` page and checks that both the book title and the names of the authors are correctly displayed on the resulting page.

When we created `CatalogController`, we specified that we want to have an action called `show` at hand. Therefore, we already have a stub method `show` in `app/controllers/catalog_controller.rb` and a pretty much empty view file `app/views/catalog/show.rhtml`. Let's now add some flesh around these bones.

Modifying the Controller

Implementing the `show` action in `CatalogController` is a simple two-liner. Add the following to `app/controllers/catalog_controller.rb`:

```
def show
  @book = Book.find(params[:id]) rescue nil
  return render(:text => "Not found", :status => 404)➥
unless @book
  @page_title = @book.title
end
```

All we do is to assign the `@book` instance variable with the book that matches the `id` we get from the browser. If the book is not found, we show a very simple 404 Not Found page. Then we put the title of the book in the `@page_title` instance variable to make it show in the layout.

Modifying the View

In the view file, we'll show the details of the book at hand (remember that the book title is shown by the layout file inside an `h1` element). Add the following to `app/views/catalog/show.rhtml`:

```
<h2>by <%= @book.authors.map{|a| a.name}.join(", ") %></h2>
<%= image_tag url_for_file_column(:book, :cover_image) ➥
    unless @book.cover_image.blank? %>
<dl>
  <dt>Price</dt>
  <dd>$<%= sprintf("%0.2f", @book.price) -%></dd>
  <dt>Page count</dt>
  <dd><%= @book.page_count -%></dd>
  <dt>Publisher</dt>
  <dd><%= @book.publisher.name %></dd>
  <dt>Blurb</dt>
  <dd><%= @book.blurb %></dd>
</dl>

<p><%= link_to "Back to Catalog", :action => "index" %></p>
```

Now the view will show the names of all the authors of a book separated by a comma. We also show the cover image of the book if one has been added, and other details of the book. We run the test again, and see that everything works just fine.

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
..
Finished in 0.231862 seconds.

2 tests, 11 assertions, 0 failures, 0 errors
```

Figure 4-1 shows a book detail page in action.

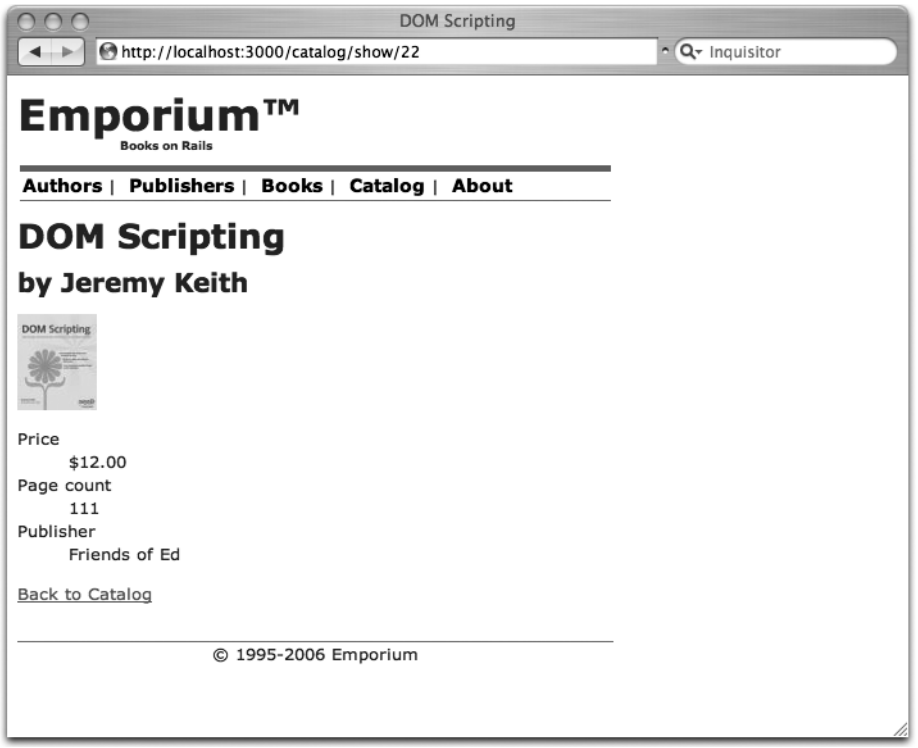

Figure 4-1. *Book detail page*

Adding Links

Now that we have pages for individual books, it would be a good idea to link to them from the catalog list page. Let's make sure that a link exists for each book on the catalog/index page. We create a separate method for checking the links, and then call that method from both the

browse_index and go_to_second_page methods. Add the following new method and calls for it to the BrowsingTestDSL module in test/integration/browsing_and_searching_test.rb:

```
module BrowsingTestDSL
  attr_writer :name

  def browse_index
    get "/catalog"

    assert_response :success
    assert_template "catalog/index"
    assert_tag      :tag => "dl", :attributes =>
                      { :id => "books" },
                    :children =>
                      { :count => 10, :only =>
                        {:tag => "dt"}}
    assert_tag :tag => "dt", :content => "The Idiot"
    check_book_links
  end

  def go_to_second_page
    get "/catalog?page=2"
    assert_response :success
    assert_template "catalog/index"
    assert_equal    Book.find_by_title("Pro Rails E-Commerce"),
                    assigns(:books).last
    check_book_links
  end

  def get_book_details_for(title)
    @book = Book.find_by_title(title)
    get "/catalog/show/#{@book.id}"
    assert_response :success
    assert_template "catalog/show"

    assert_tag      :tag => "h1",
                    :content => @book.title
    assert_tag      :tag => "h2",
                    :content => "by #{@book.authors.map{|a| a.name}}"
  end

  def check_book_links
    for book in assigns(:books)
      assert_tag    :tag => "a", :attributes =>
                      { :href => "/catalog/show/#{book.id}"}
    end
  end
end
```

The next thing to do is to create the links on the index page. Open `app/views/catalog/index.rhtml` and add the highlighted code:

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

Run the tests again. See for yourself the results in Figure 4-2, and bathe in the glory of having implemented yet another user story.

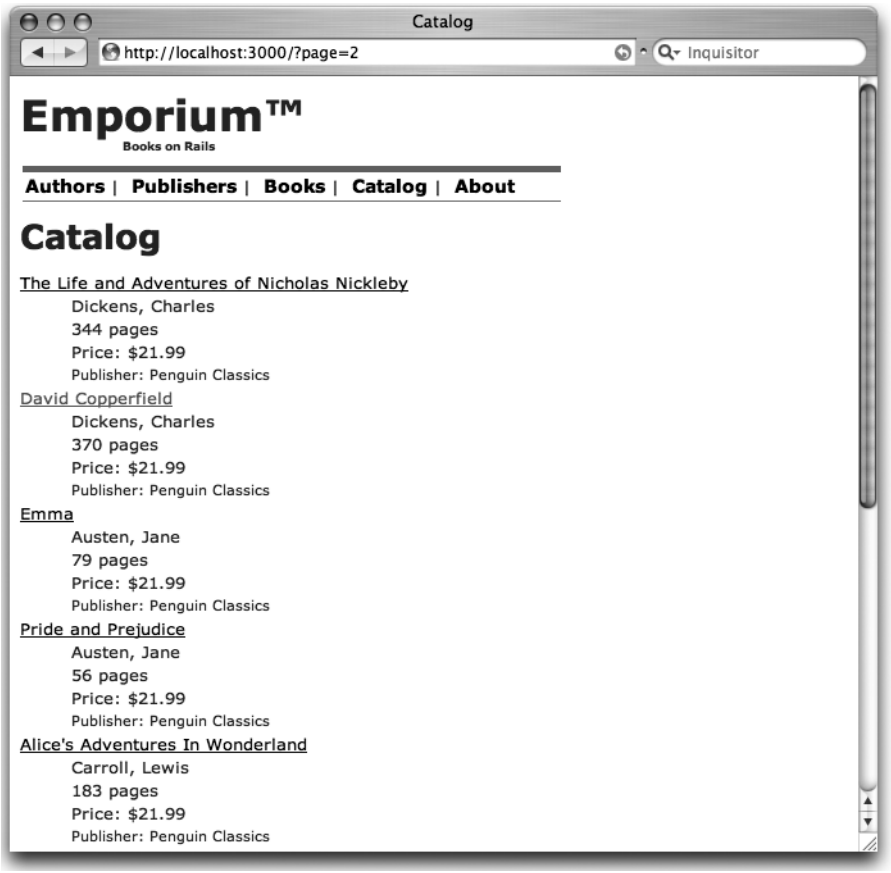

Figure 4-2. *Catalog list page with links*

Implementing the Search Books User Story

An online bookstore, or any other e-commerce site for that matter, would be nothing without search functionality. For simple cases and low loads, it would be enough to just create SQL `SELECT` queries from the search terms to find matching items. However, when the load gets higher and there is more than one table involved in the search, it is worthwhile to use a real full-text search engine for the search. In this chapter, we will use a full-text engine written in Ruby called Ferret (`http://ferret.davebalmain.com/trac`).

Using the Ferret Search Engine

Ferret is open source and uses the MIT license, so it should be a safe choice for any kind of Rails project. There are a couple of other engines available (notably Hyper Estraier and the `acts_as_searchable` Rails plugin that uses it), but we'll use Ferret in this chapter for several reasons:

- Using the `acts_as_ferret` Rails plugin makes integrating Ferret with Rails applications really simple.
- Ferret is a full port of the more famous Java search engine Apache Lucene (`http://lucene.apache.org/`), supporting its whole API. That makes Ferret an easy choice for former Java developers.
- Ferret is reasonably fast, even though it's written in a scripting language. Also, there are versions of Ferret where parts or all of the code are written in C, making it suitable for even the most challenging situations.

Installing Ferret is as easy as a single command:

```
$ sudo gem install ferret
```

The next step is to install the `acts_as_ferret` plugin. We could use Ferret directly, but why duplicate proven and tested code, especially since using the plugin also makes our own code a lot cleaner and less error-prone? You can install the plugin with the normal Rails `plugin` command:

```
$ script/plugin install ➥
svn://projects.jkraemer.net/acts_as_ferret/trunk/plugin/acts_as_ferret
```

```
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/LICENSE
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/rakefile
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/init.rb
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/lib
A    /home/george/projects/emporium/vendor/plugins/ ➥
acts_as_ferret/lib/acts_as_ferret.rb
A    /home/george/projects/emporium/vendor/plugins/acts_as_ferret/README
Exported revision 59.
```

Now that both Ferret and acts_as_ferret are installed, the only thing we need to make our books searchable is one line in app/models/book.rb:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  acts_as_ferret :fields => [:title, :author_names]
  # lots of omitted code

end
```

With that single line, we have made it possible to do fast searches on books according to their titles and authors. acts_as_ferret now intercepts all create, update, and delete operations of the Book class and updates its full-text index accordingly.

But wait a minute! There is no attribute author_names in the books table. That is correct. Fortunately, acts_as_ferret can index even objects' instance method values, so we'll add a method called author_names to the Book model class. Change app/models/book.rb as shown here:

```
class Book < ActiveRecord::Base
  has_and_belongs_to_many :authors
  belongs_to :publisher

  acts_as_ferret :fields => [:title, :author_names]
  file_column :cover_image

  validates_length_of :title, :in => 1..255
  validates_presence_of :publisher
  validates_presence_of :authors
  validates_presence_of :published_at
  validates_numericality_of :page_count, :only_integer => true
  validates_numericality_of :price
  validates_format_of :isbn, :with => /[0-9\-xX]{13}/
  validates_uniqueness_of :isbn

  def author_names
    self.authors.map do |a|
      a.name
    end.join(", ") rescue ""
  end
end
```

The author_names method iterates over all of the authors for a given book and returns their names separated by a comma. If there are no authors, it returns an empty string to avoid data type problems in the indexing code.

acts_as_ferret stores its indices in index/[environment] inside your Rails application directory, so your tests won't affect the indices used in development and production. That said, let's create a unit test for the Book class to make sure that the search works correctly. Open test/unit/book_test.rb and paste the following code after the existing tests:

```
def test_ferret
  Book.rebuild_index

  assert Book.find_by_contents("Pride and Prejudice")

  assert_difference Book, :count do
    book = Book.new(:title => 'The Success of Open Source',
              :published_at => Time.now, :page_count => 500,
              :price => 59.99, :isbn => '0-674-01292-5')
    book.authors << Author.create(:first_name => "Steven", :last_name => "Weber")
    book.publisher = Publisher.find(1)
    assert book.valid?
    book.save

    assert_equal 1, Book.find_by_contents("Open Source").size
    assert_equal 1, Book.find_by_contents("Steven Weber").size
  end
end
```

In the beginning of the test, we make sure that the Ferret index is up-to-date. Rails unit tests empty the test database before each test run, but the same doesn't hold true for the index. Therefore it's better to rebuild it so that we can be sure that we always have a similar index before we start running the tests.

Next, we use the class method Book.find_by_contents to search for a book that has "Pride and Prejudice" in either its title or authors. The result should be positive because there is a book with that name in the fixtures we created at the beginning of this chapter.

find_by_contents is a class method created automatically by acts_as_ferret. It is the workhorse of the plugin, taking as its parameters a string of search terms, and returning an array of zero or more objects, just like the normal ActiveRecord find(:all) and find_all_by_* methods.

The last part of the test case tests that a new book is correctly added to the index and is found when searched. We have put this code inside an assert_difference block, just as we did in Chapter 2, to make sure that the book is also saved to the database. We run the test and see that our search engine is working like a dream.

Now that our Book model supports fast search, it's time to implement a search interface for our bookstore.

Updating the Integration Test

We start by extending our integration test to span searching, too. We do this by adding a new method, `searches_for_tolstoy`, to the `BrowsingTestDSL` module in `test/integration/browsing_and_searching_test.rb`, as shown in Listing 4-2.

Listing 4-2. *Test Method for Book Searches*

```
require "#{File.dirname(__FILE__)}/../test_helper"

class BrowsingAndSearchingTest < ActionController::IntegrationTest
  fixtures :publishers, :authors, :books, :authors_books

  def test_browsing_the_site
    jill = enter_site(:jill)
    jill.browse_index
    jill.go_to_second_page
    jill.searches_for_tolstoy
  end

  def test_getting_details
    jill = enter_site(:jill)
    jill.get_book_details_for "Pride and Prejudice"
  end

  private

  module BrowsingTestDSL
    include ERB::Util
    attr_writer :name

    def browse_index
      get "/catalog"

      assert_response :success
      assert_template "catalog/index"
      assert_tag      :tag => "dl", :attributes =>
                        { :id => "books" },
                      :children =>
                        { :count => 10, :only =>
                          {:tag => "dt"}}
      assert_tag :tag => "dt", :content => "The Idiot"
      check_book_links
    end
```

```
  def go_to_second_page
    get "/catalog?page=2"
    assert_response :success
    assert_template "catalog/index"
    assert_equal    Book.find_by_title("Pro Rails E-Commerce"),
                    assigns(:books).last
    check_book_links
  end

  def get_book_details_for(title)
    @book = Book.find_by_title(title)

    get "/catalog/show/#{@book.id}"
    assert_response :success
    assert_template "catalog/show"

    assert_tag      :tag => "h1",
                    :content => @book.title
    assert_tag      :tag => "h2",
                    :content => "by #{@book.authors.map{|a| a.name}}"
  end

  def searches_for_tolstoy
    leo = Author.find_by_first_name_and_last_name("Leo", "Tolstoy")

    get "/catalog/search?q=#{url_encode("Leo Tolstoy")}"
    assert_response :success
    assert_template "catalog/search"

    assert_tag      :tag => "dl", :attributes =>
                      { :id => "books" },
                    :children =>
                      { :count => leo.books.size, :only =>
                        {:tag => "dt"}}

    leo.books.each do |book|
      assert_tag :tag => "dt", :content => book.title
    end
  end

  def check_book_links
    for book in assigns(:books)
      assert_tag    :tag => "a", :attributes =>
                      { :href => "/catalog/show/#{book.id}"}
    end
  end
end
```

```
  def enter_site(name)
    open_session do |session|
      session.extend(BrowsingTestDSL)
      session.name = name
      yield session if block_given?
    end
  end
end
```

Our new test method makes Jill search for Leo Tolstoy with a search form and checks that the resulting result list will have exactly as many books as Leo has provided the shop, namely two. We use the `url_encode` method to escape white space from the search string. It is provided by the `ERB::Util` library, so we need to require it at the beginning of our module. Last, we test that the books in the resulting list have the correct names by going through all the books written by Leo and checking that there is a `dt` element containing the book's title.

Creating a Search Form Template

Now that we have the integration test made, we can start implementing the thing for real. We first create a simple search form template. Save the following code in `app/views/catalog/_search_box.rhtml`:

```
<%= form_tag({:action => "search"}, {:method => "get"}) %>
<%= text_field_tag :q %>
<%= submit_tag "Search" %>
<%= end_form_tag %>
```

Saving it as a partial makes it possible for us to easily embed the search form in other pages.

In the code, we create a simple form that points to the `search` action and uses the `get` method. Using `get` instead of `post` will make the query string be a part of the URL. That way, Jill can circulate a link to her search results to all of her friends. Our form has only two elements: a text field `q` and the submit button.

In the actual search template, we display the partial using the `render` method. Save the following line to `app/views/catalog/search.rhtml`:

```
<%= render :partial => "search_box" %>
```

Modifying the Controller

Next, open app/controllers/catalog_controller.rb and implement the search action.

```
def search
  @page_title = "Search"
  if params[:commit] == "Search" || params[:q]
    @books = Book.find_by_contents(params[:q].to_s.upcase)
    unless @books.size > 0
      flash.now[:notice] = "No books found matching your criteria"
    end
  end
end
```

In the search action, we first specify the title for the page. Then we continue in two different directions, depending on whether the search form was already submitted or the search page was just requested normally. We do the separation by checking if either the value of a query parameter commit is "Search" or the query variable q is specified. From the _search.rhtml partial view, q contains the search text that was submitted by the search form.

If our code determines that the form has been submitted, it executes the search using the find_by_contents class method and the query parameter q. Furthermore, if there are no books found with the terms, it sets the flash notice to show a message to the user.

Modifying the View

Now we need to extend our search view so that it shows either the books found or the "Not found" notice. Add the following to app/views/catalog/search.rhtml:

```
<%= render :partial => "search_box" %>

<% if @books %>
<p>Your search "<%= params[:q] %>" produced
<%= pluralize @books.size, "result" %>:</p>
<%= render(:partial => "books") %>
<% end %>
```

If the search was successful, we also tell how many hits there were. We use the pluralize helper to show the number of books, and the word "result" in singular or plural depending on the count. Last, we render a partial to show a list of matching books.

We don't have a partial view called books yet, so we need to create it. In the `index` action, we also showed a list of books, so it is a good place to extract the list. Move the following code from `app/views/catalog/index.rhtml` to `app/views/catalog/_books.rhtml`.

```
<dl id="books">
  <% for book in @books %>
    <dt><%= link_to book.title.t, :action => "show", :id => book %></dt>
    <% for author in book.authors %>
      <dd><%= author.last_name %>, <%= author.first_name %></dd>
    <% end %>
      <dd><%= pluralize(book.page_count, "page") %></dd>
      <dd>Price: $<%= sprintf("%.2f", book.price) %></dd>
      <dd><small>Publisher: <%= book.publisher.name %></small></dd>
  <% end %>
</dl>
```

Now we can just replace the moved code in `index.rhtml` with a similar `render` call that we have in the end of the `search.rhtml` template, and that's it! We have a functioning search form in the bookstore.

If you have already added some books to your development system, you can point your browser to `/catalog/search` on your development site and see the result for yourself, as shown in Figure 4-3. (First, you will need to restart your web server, so it will pick up the introduced Ferret code.)

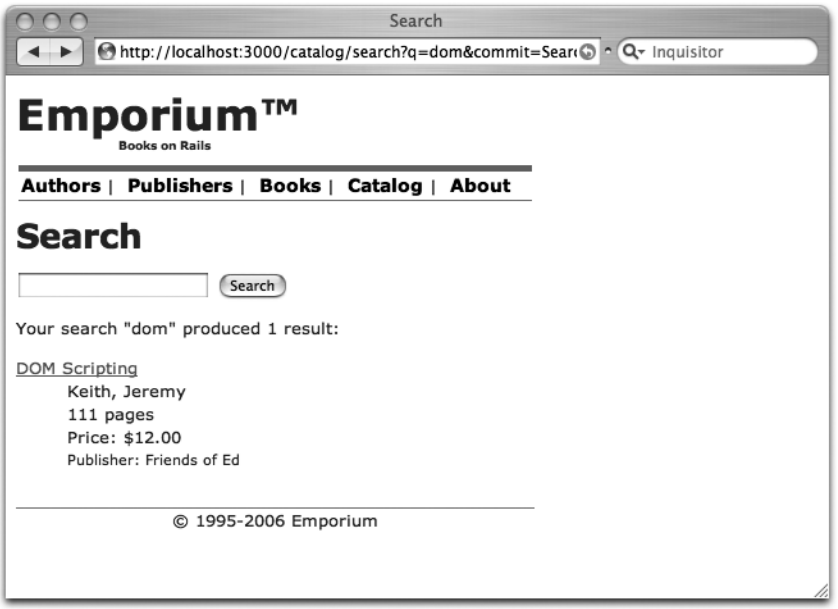

Figure 4-3. *Search interface*

We also need a link to our search functionality, so add the following to `app/views/catalog/index.rhtml`:

```
<p><%= link_to "Search", :action => "search" %></p>

<%= render(:partial => "books") %>

<%= link_to 'Previous page'.t, { :page => @book_pages.current.previous } if ➥
@book_pages.current.previous %>
<%= link_to 'Next page'.t, { :page => @book_pages.current.next } if ➥
@book_pages.current.next %>
```

The search functionality is now implemented

Implementing the Get Latest Books User Story

So far, we have created a book catalog that lets Jill browse and search books, and see their details. The last part of the sprint is to implement the ultimate desire of a book-lover: a list of the latest books. We'll implement this feature both as a normal web page and as an RSS feed, so that Jill can skip the step of using a browser altogether. Again, we'll start by writing a test for the latest books page.

Updating the Integration Test

Add another method to the `BrowsingTestDSL` module in `test/integration/browsing_and_searching_test.rb`:

```
def views_latest_books
  get "/catalog/latest"
  assert_response :success
  assert_template "catalog/latest"

  assert_tag      :tag => "dl", :attributes =>
                    { :id => "books" },
                  :children =>
                    { :count => 10, :only =>
                      {:tag => "dt"}}
  Book.latest.each do |book|
    assert_tag    :tag => "dt", :content => book.title
  end
  check_book_links
end
```

You can see that the method is similar to `browse_index` and `go_to_second_page`, but it has a different URL and desired template. The only thing special here is that we iterate over the `Book` objects returned by `Book.latest` and check that there is a `dt` element for each book. To make this work, we first need to create a `latest` class method for our `Book` class. Add the following method to `app/models/book.rb`:

```
def self.latest
  find :all, :limit => 10, :order => "books.id desc",
             :include => [:authors, :publisher]
end
```

We could have used the `find` method as such in our test. However, we're going to need the exact same code later, so it's a good idea to wrap it inside a class method. We also need to add a call to our new method in the actual test case:

```
def test_browsing_the_site
  jill = enter_site(:jill)
  jill.browse_index
  jill.go_to_second_page
  jill.get_book_details_for "Pride and Prejudice"
  jill.searches_for_tolstoy
  jill.views_latest_books
end
```

Now that we have a (failing, but you guessed that) test in place, the next thing to do is to update the controller.

Modifying the Controller

Open `app/controllers/catalog_controller.rb` and fill the `latest` action with content:

```
def latest
  @page_title = "Latest Books"
  @books = Book.latest
end
```

There's nothing special in there. We just set the page title and then use the `Book.latest` class method we just created to fetch the ten latest books.

Modifying the View

The view file, `app/views/catalog/latest.rhtml`, is even simpler:

```
<%= render :partial => "books" %>
```

We can fire our test case and see that everything works oh so smoothly.

```
$ ruby test/integration/browsing_and_searching_test.rb
```

```
Loaded suite test/integration/browsing_and_searching_test
Started
..
Finished in 0.478978 seconds.

2 tests, 56 assertions, 0 failures, 0 errors
```

We double-check in the browser to see the page shown in Figure 4-4. Filled with self-confidence, we rush on to the final task of this code sprint.

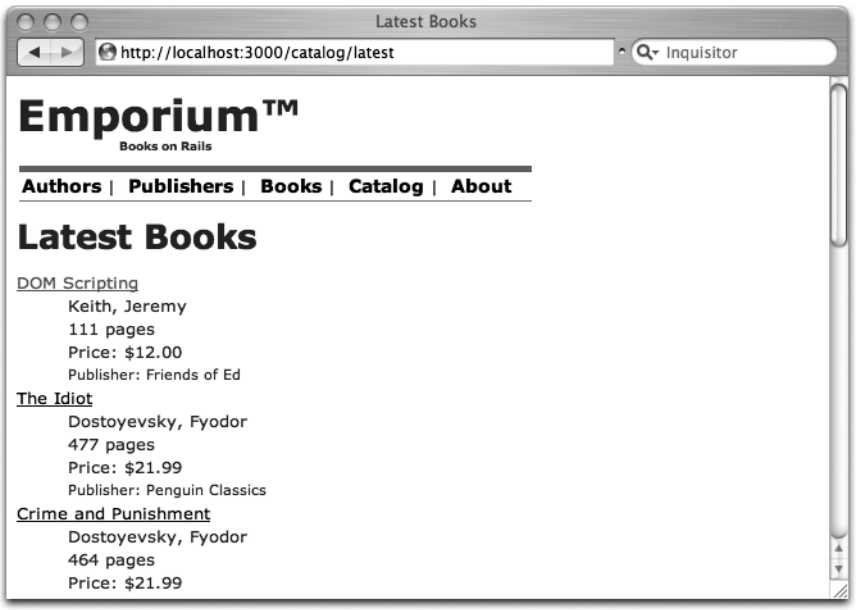

Figure 4-4. *Latest books page*

Creating an RSS Feed

Creating an RSS feed in Rails is painstakingly easy. RSS feeds are essentially just XML files served like normal HTML pages. Rails supports three kinds of template files out of the box. You are already familiar with the HTML templates with the `.rhtml` suffix. The second type is the Builder XML template, with an `.rxml` suffix, which we will use for this case. The third type? You will learn about that in the next chapter, and boy will that be fun. But first we'll create an RSS feed for Jill.

Once more, we'll create another method in our integration test. Add the following method to the `BrowsingTestDSL` module in `test/integration/browsing_and_searching_test.rb`:

```
def reads_rss
  get "/catalog/rss"
  assert_response :success
  assert_template "catalog/rss"
  assert_equal "application/xml", response.headers["type"]

  assert_tag      :tag => "channel",
                  :children =>
                    { :count => 10, :only =>
                      {:tag => "item"}}
  Book.latest.each do |book|
    assert_tag    :tag => "title", :content => book.title
  end
end
```

The method follows the familiar scheme. However, this time we also check that the response type is XML instead of HTML. It is also worth noting that we can use the same `assert_tag` methods here that we used for HTML documents, even though the output is XML.

Just as before, we call our new method from the `test_browsing_the_site` test method:

```
def test_browsing_the_site
  jill = enter_site(:jill)
  jill.browse_index
  jill.go_to_second_page
  jill.get_book_details_for "Pride and Prejudice"
  jill.searches_for_tolstoy
  jill.views_latest_books
  jill.reads_rss
end
```

Implementing the controller method is straightforward. We use the same information as in the `latest` action, so we can call it the same way we call any other method. After that, we just render our action normally, only this time, we don't want to use the layout file (because we're rendering XML).

```
def rss
  latest
  render :layout => false
end
```

The view file is where the most difference between a normal HTML page and a Rails-powered RSS feed lies. This time, we don't use the standard `.rhtml` templates, but rather `.rxml` templates powered by the Builder library. With Builder, XML output is specified using nested code blocks. For our RSS feed, we'll create the `app/views/catalog/rss.rxml` file, as shown in Listing 4-3.

Listing 4-3. *app/views/catalog/rss.rxml*

```
xml.instruct! :xml, :version=>"1.0", :encoding=>"UTF-8"

xml.rss("version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/") do
  xml.channel do
    xml.title @page_title
    xml.link(url_for(:action => "index", :only_path => false))
    xml.language "en-us"
    xml.ttl "40"
    xml.description "Emporium: Books for people"

    for book in @books
      xml.item do
        xml.title(book.title)
        xml.description("#{book.title} by #{book.author_names}")
        xml.pubDate(book.created_at.to_s(:long))
        xml.guid(url_for(:action => "show", :id => book, :only_path => false))
        xml.link(url_for(:action => "show", :id => book, :only_path => false))
      end
    end
  end
end
```

Every code block started by an `xml.tag` command in a Builder template will result in a `<tag>` element in the output. Thus, the output of the code in Listing 4-3 would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>Latest Books</title>
    <link>http://0.0.0.0:3000/catalog</link>
    <language>en-us</language>
    <ttl>40</ttl>
    <description>Emporium: Books for people</description>

    <item>
      <title>The Idiot</title>
      <description>The Idiot by Fyodor Dostoyevsky</description>
      <pubDate>April 26, 2006 20:18</pubDate>
      <guid>http://0.0.0.0:3000/catalog/show/17</guid>
      <link>http://0.0.0.0:3000/catalog/show/17</link>
    </item>

  ... more items ...

  </channel>
</rss>
```

Note that we can use all the normal Rails helper methods, like `url_for`, in `.rxml` templates, just as in normal `.rhtml` views. However, because we're not creating the XML code by hand, we can be sure that the output is always well-formed XML.

Running the integration test reveals that everything works fine. Encouraged, we open `http://localhost:3000/catalog/rss` in a browser that supports RSS feeds (such as Safari on Mac OS X or Firefox on other platforms) and show George how the feed functionality works for Jill, as shown in Figure 4-5. George is excited, and we can pat ourselves on the back. Another sprint is completed.

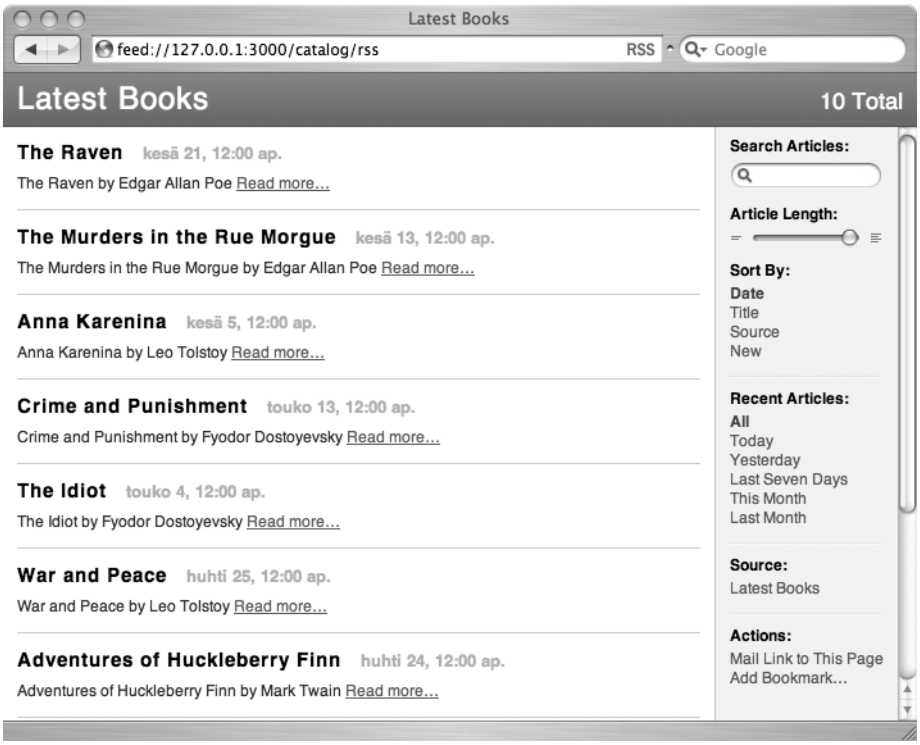

Figure 4-5. *Working RSS feed*

Summary

In this chapter, we implemented the basic functionality of the online bookstore that is visible to a normal user like Jill. This consisted of four user stories: browsing the list of books, searching books, visiting pages for individual books, and seeing lists of latest books in the store in both a web page and an RSS form.

During the course of the chapter, we showed you how to use the `include` parameter in ActiveRecord finder methods to avoid unnecessary SQL queries and use layouts to avoid repeating view code. We also integrated the Ferret full-text search engine with our Rails application using the acts_as_ferret plugin. Finally, we created RSS feeds using Builder XML templates, which saved us a lot of time. In the next chapter we will create a shopping cart for Jill to fill.