Extracted from:

Rails for Java Developers

Chapter 5

Coordinating Activities with ActionController

Controllers coordinate the activities of views and models in the MVC paradigm. Controllers are responsible for the following:

- Collecting input from the user
- Creating model objects to handle the user's request
- Selecting the appropriate view code to render

Along the way, controllers are responsible for logic that is associated with the user request (as opposed to with a specific model object). Such logic includes the following:

- Authentication and authorization
- Business rules that involve multiple model objects
- Auditing
- Error handling

In addition to these responsibilities, most web application frameworks give controllers a web-specific responsibility as well. Web controllers provide an object model wrapper for the idioms of the Web: URLs, HTTP requests, headers, cookies, and so on. At the controller level, web applications are explicitly *web* programming. (By contrast, the model layer code is much more likely to be reusable outside of a web app.) In Rails, the ActionController library implements the controller layer. In this chapter, we will introduce ActionController by comparing it to a Struts application. We will start with basic CRUD and then drill in to more advanced issues such as session management, filters, and caching.

5.1 Routing Basics: From URL to Controller+Method

To access a web application, you need a URL. For our Struts sample application, the people list view lives at /appfuse_people/editPerson.html?method=Search. How does this URL get routed to running code in a Java web application? Typically, the first part of the name (appfuse_people) identifies a .war file or directory on the server that corresponds to a particular web application. Java applications often include an Ant task to copy the application code and resources to the appropriate directory on the server.

Download code/appfuse_people/build.xml

```
<target name="deploy-web" depends="compile-jsp" if="tomcat.home"
    description="deploy only web classes to servlet container's deploy directory">

    <echo message="Deploying web application to ${tomcat.home}/webapps"/>
    <copy todir="${tomcat.home}/webapps/${webapp.name}">
        <fileset dir="${webapp.target}"
            excludes="**/web-test.xml,**/web.xml,**/*-resources.xml"/>
    </copy>

</target>
```

For a Struts application, the next part of the name (editPerson.html) is pattern matched to the Struts ActionServlet via a servlet and servlet-mapping elements in web.xml. Many Struts applications use the distinctive .do suffix; in our example, we have followed AppFuse's lead in simply using .html:

Download code/appfuse_people/web/WEB-INF/web.xml

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

These two steps do not exist in Rails development. Rails does not run more than one web application within a process—if you want multiple web applications, you run them in separate processes. Since *all* Rails code is routed to the ActionController layer, you don't have to take a separate configuration step to specify "I want to use ActionController." Rails applications also do not copy files into the web server during

development. During development, Rails code is written and executed in a single directory tree. This is part of the reason that Rails application development is so interactive: changes take effect immediately, without a deploy step.

Most Java developers find ways to simplify these two steps. Frameworks such as AppFuse create the appropriate build.xml and web.xml settings for you. Inspired in part by Rails, many Java developers now run their development code from the same directory, avoiding part of the overhead of the compile/deploy cycle.

The more important part of routing happens within the Struts ActionServlet and Rails ActionController. Struts uses settings in struts-config.xml to convert editPerson.html?method=Search into a method call:

```
<action
  path="/editPerson"
  type="com.relevancellc.people.webapp.action.PersonAction" ...
```

The path attribute matches editPerson to the class named by the type attribute: PersonAction. Finally, the query string ?method=Search leads us to the search method on PersonAction.

The Rails URL for the people list view is /people/list. Just as with Struts, Rails uses routing to convert this URL into a method on an object. In Rails, the routing is described not with XML but with Ruby code. Here is a simple routing file:

Download code/people/config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id'
end
```

The :controller portion of the route maps the first portion of the URL to a controller class. A standard convention for capitalization and class naming is used, so people becomes PeopleController. The mechanism is general, so this routing entry also implies that a URL that begins with foo will attempt to find a (nonexistent in this case) FooController. The :action portion of the route maps the second location component to a method. So, list invokes the list method. Again, the mechanism is general, so /people/foo would attempt to find a nonexistent foo method on the PeopleController. Finally, the :id maps to an id parameter, which is optional. In methods such as create and update that need an object to operate on, the id is conventionally a primary key.

Many opponents of Rails have criticized this default routing because they do not like the implied naming scheme. This entirely misses the point. Rails default routing makes *trivial things trivial*. It is easy to bring up a Rails server with a bunch of controllers that use this default route. The design philosophy is "pay as you go." The default routing gives you something simple, generic, and free. If you want more control, you can have that too, but you have to write some routing configuration, just as you do in Struts. You will see more advanced routing in Section 5.6, *Routing in Depth*, on page 153.

5.2 List and Show Actions: The R in CRUD

Now that we can route from URLs to code, let's look at the code. In our Struts application, /appfuse_people/editPerson.html?method=Search takes us to the search method of PersonAction:

Download code/appfuse_people/src/web/com/relevancellc/people/webapp/action/PersonAction.java

```
public ActionForward search(ActionMapping mapping, ActionForm form,
                            HttpServletRequest request,
                            HttpServletResponse response)
    throws Exception {
  PersonManager mgr = (PersonManager) getBean("personManager");
  List people = mgr.getPeople(null);
  request.setAttribute(Constants.PERSON_LIST, people);
  return mapping.findForward("list");
}
```

The signature of the method contains specific parameters for accessing the web object model (request and response) and the Struts object model (mapping and form). The object model is then used to load the people, and forward to the view, through the following steps:

1. On line 5, we look up the manager object that will actually do the work.
2. On line 6, we get the people object that will be rendered in the view.
3. On line 7, we add the people to the request, which makes the people available to the view.
4. Finally on line 8, we select the view that should render the list.

Behind the scenes is a lot of layering. The manager in its turn delegates to a DAO, which actually does the data access. The manager and DAO layers require two Java source files each: an interface to the layer and at

least one implementation. In addition, the connections between layers are configured using Spring Dependency Injection. At the end of the chain, here is the code that does the work:

Download code/appfuse_people/src/dao/com/relevancellc/people/dao/hibernate/PersonDaoHibernate.java

```
public List getPeople(Person person) {
  return getHibernateTemplate().find("from Person");
}
```

If you understand how this all works in Struts, the transition to Rails is straightforward. A typical Rails controller does the same steps. This is not obvious at first, because at every step, the Rails approach makes a different stylistic choice. Here is the code:

Download code/people/app/controllers/people_controller.rb

```
def list
  @search = params[:search]
  if @search.blank?
    @person_pages, @people = paginate :people, :per_page => 10
  else
    query = ['first_name = :search or last_name = :search',
            {:search=>@search}]
    @person_pages, @people = paginate :people,
                             :per_page => 10, :conditions=>query
  end
end
```

The Rails list has no parameters! Of course, the same kinds of information are available. The difference is that the request and response objects are member variables (with accessor methods) on the controller. The Java philosophy here is "Explicit is better. It is easy to read a Struts action and see what objects you should be working with." The Rails philosophy is "Implicit is better, at least for things that are common. This is a web app, so requests and responses are pretty common! Learn them once, and never have to type or read them again."

The Rails list does not delegate to intermediate layers. There is no manager or DAO layer, just a call to paginate, which in turn directly accesses ActiveRecord. This is certainly an important difference, and we want to be careful in laying out why we think both the Java and Rails strategies make sense. Imagine the following conversation between Rita the Rails developer and Jim the Java developer:

Rita: *Why do you bother with all those layers?*

Jim: *The layers make it easier to test the code and to reuse the code in different contexts. For example, the manager layer has no web depen-*

dencies, so that code can be reused in a Swing application or over an RMI connection.

Rita: *Still, it must take forever to write all that extra code.*

Jim: *It isn't so bad. We have much more elaborate IDE support in the Java world. Plus, tools such as AppFuse or Maven can be used to do a lot of the boilerplate work. Aren't you worried that your Rails app is a dead end and that your code is inflexible and untestable?*

Rita: *Not at all. I am building the layers I need right now. If I need more layers later, it is much easier to add them. Dynamic typing makes it much easier to plug in new code or execute the existing code in a new context.*

Jim: *But with dynamic typing, how do you make sure your code works? I am used to the compiler making sure that variables are of the correct type.*

Rita: *We validate our code with unit tests, functional tests, integration tests, black-box tests, code reviews, and code coverage. Do you do the same?*

Jim: *You bet!*

In short, the Java approach (lots of layers, dependency injection, good tooling) is a reasonable response to Java's class-centric, statically typed object model. The Ruby approach (layers on demand, less tooling) is a reasonable approach to Ruby's object-centric, dynamically typed object model.

The Rails list method creates person_pages and people variables, but it does nothing to make these variables available to the view. Again, the difference is that Rails does things *implicitly*. When you create instance variables in a controller method, they are automatically copied into the view using reflection. This approach takes advantage of the fact that Ruby classes are open, and this approach can pick up arbitrary variables at any time.

Finally, the Rails code does not appear to select a view to render. Again, this is because Rails provides an implicit default behavior. When you exit a controller method, the default behavior is to render a view template file named app/views/{controllername}/{methodname}.rhtml. As you will see next, Rails provides a render method that you can use to override this behavior.

Now that you have seen the list action, you will look at the code for showing an edit form for a single person. Our Struts implementation uses a single action named edit for both the "new" and "update" varieties:

Download code/appfuse_people/src/web/com/relevancellc/people/webapp/action/PersonAction.java

```
public ActionForward edit(ActionMapping mapping, ActionForm form,
                          HttpServletRequest request,
                          HttpServletResponse response)
    throws Exception {
  PersonForm personForm = (PersonForm) form;
  if (personForm.getId() != null) {
    PersonManager mgr = (PersonManager) getBean("personManager");
    Person person = mgr.getPerson(personForm.getId());
    personForm = (PersonForm) convert(person);
    updateFormBean(mapping, request, personForm);
  }
  return mapping.findForward("edit");
}
```

This code goes through the same series of steps you saw earlier: Call into another layer to get the object, put the object into request scope, and select the mapping to the view. The novel part is interacting with the form bean. The form is an instance of PersonForm. The form bean represents the web form data associated with a person. Because the form is functionally a subset of a Person model, the form bean class can be autogenerated. You can accomplish this with an XDoclet tag at the top of the Person class:

```
@struts.form include-all="true" extends="BaseForm"
```

To display an edit form, the edit action needs to copy data from the model person to its form representation. The convert method does this. You could write individual convert methods for each model/form pair in an application. A far simpler approach is to use JavaBean introspection to write a generic convert method. Our approach uses a generic convert method that is included in AppFuse.

The Rails equivalent uses two actions: new and edit:

Download code/people/app/controllers/people_controller.rb

```
def edit
  @person = Person.find(params[:id])
end

def new
  @person = Person.new
end
```

The Rails version does the same things but in a different way. In Rails applications, there is no distinction between model objects and form beans; ActiveRecord objects serve both purposes. As a result, there is no form argument or convert step. The Rails version has two methods because Rails applications typically render "new" and "edit" with two different templates. (This is not as redundant as it sounds; the two templates delegate to a single partial template that actually draws the form.)

5.3 Create, Update, and Delete Actions

Create, update, and delete actions tend to have more interesting code because they alter state. As a result, they have to deal with validation, status messages, and redirection. Here is a Struts action method that will save or update a person:

Download code/appfuse_people/src/web/com/relevancellc/people/webapp/action/PersonAction.java

```
public ActionForward save(ActionMapping mapping, ActionForm form,
                          HttpServletRequest request,
                          HttpServletResponse response)
    throws Exception {
  ActionMessages messages = new ActionMessages();
  PersonForm personForm = (PersonForm) form;
  boolean isNew = ("".equals(personForm.getId()));
  PersonManager mgr = (PersonManager) getBean("personManager");
  Person person = (Person) convert(personForm);
  mgr.savePerson(person);
  if (isNew) {
    messages.add(ActionMessages.GLOBAL_MESSAGE,
        new ActionMessage("person.added"));
    saveMessages(request.getSession(), messages);
    return mapping.findForward("mainMenu");
  } else {
    messages.add(ActionMessages.GLOBAL_MESSAGE,
        new ActionMessage("person.updated"));
    saveMessages(request, messages);
    return mapping.findForward("viewPeople");
  }
}
```

Let's begin by considering the happy case where the user's edits are successful. Much of this code is similar to previous examples; the new part is the addition of a status message. In line 5 we create an ActionMessages instance to hold a status message, and in lines 12–14 and 17–19 we save the ActionMessages into the request so they can be rendered in the view.

Here is the Rails version of update:

Download code/people/app/controllers/people_controller.rb

```
def update
  @person = Person.find(params[:id])
  if @person.update_attributes(params[:person])
    flash[:notice] = 'Person was successfully updated.'
    redirect_to :action => 'show', :id => @person
  else
    render :action => 'edit'
  end
end
```

The actual update happens on line 3. update_attributes is an ActiveRecord method that sets multiple attributes all at once. Like its cousins create and save, update_attributes automatically performs validations. Since the params[:person] hash contains all the name/value pairs from the input form, a single call to update_attributes does everything necessary to update the @person instance.

Like the Struts update, the Rails version of update sets a status message. In line 4, the message "Person was successfully updated." is added to a special object called the *flash*. The flash is designed to deal with the fact that updates are generally followed by redirects.

flash

So, saving a status into a member variable does no good—after the redirect, the status variable will be lost. Saving into the session instead will work, but then you have to remember to remove the status message from the session later. And that is exactly what the flash does: saves an object into the session and then automatically removes the status message after the next redirect.

The flash is a clever trick. Unfortunately, the data that is typically put into the flash is not clever at all. Out of the box, Rails does not support internationalization, and status messages are stored directly as strings (usually in English).

Contrast this with the Struts application, which stores keys such as "person.added." The view can later use these keys to look up an appropriately localized string. The lack of internationalization support is one of the big missing pieces in Rails. If your application needs internationalization, you will have to roll your own or use a third-party library.

After a successful update operation, the controller should redirect to a URL that does a read operation. This makes it less likely that a user will bookmark a URL that does an update, which will lead to odd results

later. Some possible choices are a show view of the object just edited, a list view of similar objects, or a top-level view. The Struts version does the redirect by calling findForward:

```
return mapping.findForward("mainMenu");
```

To verify that this forward does a redirect, you can consult the struts.xml configuration file. Everything looks good:

```
<global-forwards>
  <forward name="mainMenu" path="/mainMenu.html" redirect="true"/>
  <!-- etc. -->
</global-forwards>
```

Where Struts uses findForward for both renders and redirects, Rails has two separate methods. After a save, the controller issues an explicit redirect:

```
redirect_to :action => 'show', :id => @person
```

Notice that the redirect is named in terms of actions and parameters. Rails runs its routing table "backward" to convert from actions and parameters back into a URL. When using default routes, this URL will be /people/show/(some_int).

Now that you have seen a successful update, we'll show the case where the update fails. Both Struts and Rails provide mechanisms to validate user input.

In Struts, the Validator object automatically validates form beans, based on declarative settings in an XML file. Validations are associated with the form. To specify that the first name is required, you can use XML like this:

Download code/appfuse_people/snippets/person_form.xml

```
<form name="personForm">
  <field property="firstName" depends="required">
      <arg0 key="personForm.firstName"/>
  </field>
  <!-- other fields -->
</form>
```

The original intention of the discrete validation language was separation of concerns. Sometimes it is more convenient to keep related concerns together. Instead of writing the validation.xml file by hand, we generate the validations with XDoclet annotations in the Person model class in this way:

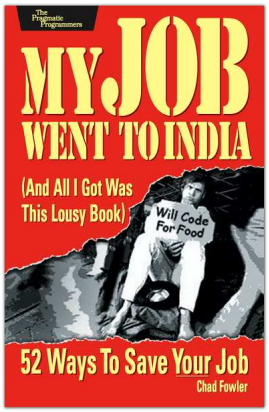
A Pragmatic Career

Welcome to the Pragmatic Community. We hope you've enjoyed this title.

Interested in improving your career? Want to make yourself more valuable to your organization, and avoid being outsourced? Then read *My Job Went to India*, and find out great ways to keep yours. If you're interested in moving your career more towards a team lead or mangement position, then read what happens *Behind Closed Doors*.

My Job Went to India

The job market is shifting. Your current job may be outsourced, perhaps to India or eastern Europe. But you can save your job and improve your career by following these practical and timely tips. See how to: • treat your career as a business • build your own brand as a software developer • develop a structured plan for keeping your skills up to date • market yourself to your company and rest of the industry • keep your job!

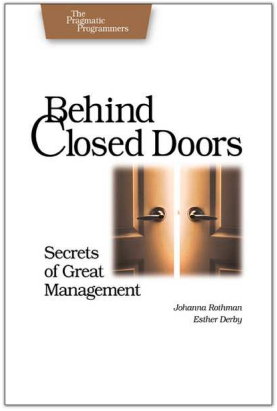


My Job Went to India: 52 Ways to Save Your Job
Chad Fowler
(185 pages) ISBN: 0-9766940-1-8. $19.95
http://pragmaticprogrammer.com/titles/mjwti

Behind Closed Doors

You can learn to be a better manager—even a great manager—with this guide. You'll find powerful tips covering:

• Delegating effectively • Using feedback and goal-setting • Developing influence • Handling one-on-one meetings • Coaching and mentoring • Deciding what work to do-and what not to do • . . . and more!



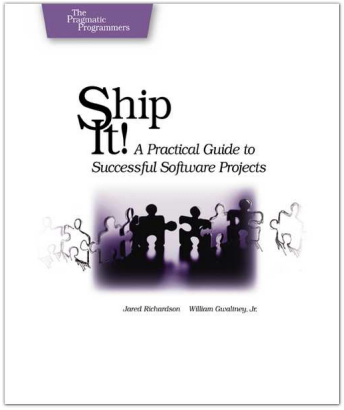
Behind Closed Doors Secrets of Great Management
Johanna Rothman and Esther Derby
(192 pages) ISBN: 0-9766940-2-6. $24.95
http://pragmaticprogrammer.com/titles/rdbcd

Pragmatic Methodology

Need to get software out the door? Then you want to see how to *Ship It!* with less fuss and more features. And every developer can benefit from the *Practices of an Agile Developer*.

Ship It!

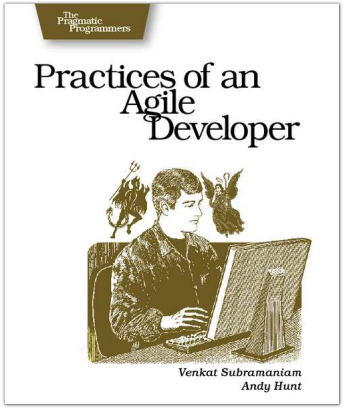
Page after page of solid advice, all tried and tested in the real world. This book offers a collection of tips that show you what tools a successful team has to use, and how to use them well. You'll get quick, easy-to-follow advice on modern techniques and when they should be applied. **You need this book if:** • You're frustrated at lack of progress on your project. • You want to make yourself and your team more valuable. • You've looked at methodologies such as Extreme Programming (XP) and felt they were too, well, extreme. • You've looked at the Rational Unified Process (RUP) or CMM/I methods and cringed at the learning curve and costs. • **You need to get software out the door without excuses**



Ship It! A Practical Guide to Successful Software Projects
Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. $29.95
http://pragmaticprogrammer.com/titles/prj

Practices of an Agile Developer

Agility is all about using feedback to respond to change. Learn how to apply the principles of agility throughout the software development process • Establish and maintain an agile working environment • Deliver what users really want • Use personal agile techniques for better coding and debugging • Use effective collaborative techniques for better teamwork • Move to an agile approach



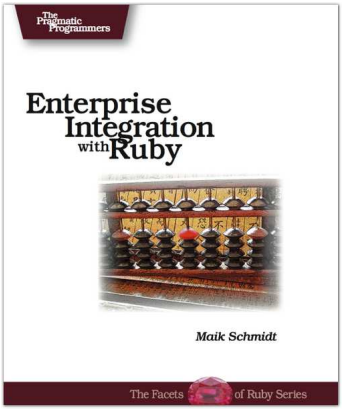
Practices of an Agile Developer: Working in the Real World
Venkat Subramaniam and Andy Hunt
(189 pages) ISBN: 0-9745140-8-X. $29.95
http://pragmaticprogrammer.com/titles/pad

Facets of Ruby Series

See how to integrate Ruby with all varieties of today's technology in *Enterprise Integration with Ruby*. And speaking of today's finest, you'll need a good text editor, too. On the Mac, we recommend TextMate.

Enterprise Integration with Ruby

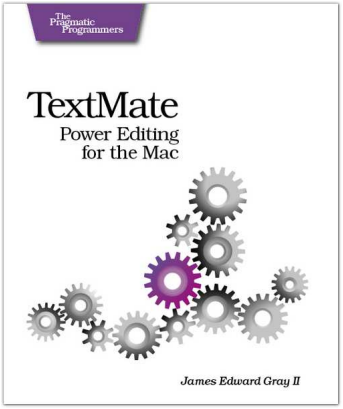
See how to use the power of Ruby to integrate all the applications in your environment. Lean how to • use relational databases directly, and via mapping layers such as ActiveRecord • Harness the power of directory services • Create, validate, and read XML documents for easy information interchange • Use both high- and low-level protocols to knit applications together



Enterprise Integration with Ruby
Maik Schmidt
(360 pages) ISBN: 0-9766940-6-9. $32.95
http://pragmaticprogrammer.com/titles/fr_eir

TextMate

If you're coding Ruby or Rails on a Mac, then you owe it to yourself to get the TextMate editor. And, once you're using TextMate, you owe it to yourself to pick up this book. It's packed with information which will help you automate all your editing tasks, saving you time to concentrate on the important stuff. Use snippets to insert boilerplate code and refactorings to move stuff around. Learn how to write your own extensions to customize it to the way you work.



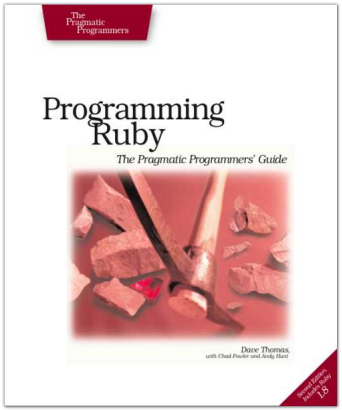
TextMate: Power Editing for the Mac
James Edward Gray II
(200 pages) ISBN: 0-9787392-3-X. $29.95
http://pragmaticprogrammer.com/titles/textmate

Facets of Ruby Series

If you're serious about Ruby, you need the definitive reference to the language. The Pickaxe: *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition*. This is *the* definitive guide for all Ruby programmers. For Rails, we have the definitive reference guide as well: the award-winning and best-selling *Agile Web Development with Rails*.

Programming Ruby (The Pickaxe)

The Pickaxe book, named for the tool on the cover, is the definitive reference to this highly-regarded language. • Up-to-date and expanded for Ruby version 1.8 • Complete documentation of all the built-in classes, modules, and methods • Complete descriptions of all ninety-eight standard libraries • 200+ pages of new content in this edition • Learn more about Ruby's web tools, unit testing, and programming philosophy

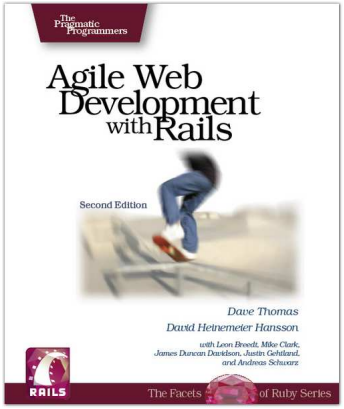


Programming Ruby: The Pragmatic Programmer's Guide, 2nd Edition
Dave Thomas with Chad Fowler and Andy Hunt
(864 pages) ISBN: 0-9745140-5-5. $44.95
http://pragmaticprogrammer.com/titles/ruby

Agile Web Development with Rails

Rails is a full-stack, open-source web framework, with integrated support for unit, functional, and integration testing. It enforces good design principles, consistency of code across your team (and across your organization), and proper release management. This is newly updated Second Edition, which goes beyond the Jolt-award winning first edition with new material on:

• Migrations • RJS templates • Respond_to • Integration Tests • Additional ActiveRecord features • Another year's worth of Rails best practices



Agile Web Development with Rails: Second Edition
Dave Thomas, and David Heinemeier Hansson with Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz
(750 pages) ISBN: 0-9776166-3-0. $39.95
http://pragmaticprogrammer.com/titles/rails2

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 programmers stay on top of their game.

Visit Us Online

Rails for Java Developers Home Page
http://pragmaticprogrammer.com/titles/fr_r4j
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: http://pragmaticprogrammer.com/titles/fr_r4j.

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