Practical Rails Plugins

ISBN-13 (pbk): 978-1-59059-993-8

ISBN-10 (pbk): 1-59059-993-4

ISBN-13 (electronic): 978-1-4302-0654-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Clay Andres
Technical Reviewers: Justin Blake, Josh Martin
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Nancy Bell
Indexer: Carol Burbo
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

PART 1

Introduction to Plugins for Ruby on Rails

In this first part of the book, we'll introduce you to the fundamentals of plugins: what they are, how they work, and how you can get started using them. Plugins come in a variety of forms, but almost all of them are installed and managed in just a few different ways, which we'll cover in Chapter 1. We'll also show you two concrete examples of plugins.

CHAPTER 1

The Power of Rails Plugins

Ruby on Rails is a powerful framework for building web applications. We won't waste precious space extolling its many virtues; if you weren't already aware of them, you probably wouldn't be reading this book. To put it succinctly, Rails makes good development practices easy and enjoyable, and allows you to build solid, well-structured, and (gasp!) maintainable applications quickly.

As most seasoned developers have learned through experience, a framework is only as good as it is extensible, and yet there's a certain trick to keeping such a thing simple and unencumbered. In fact, you might even say that these goals of simplicity and extensibility are directly at odds most of the time. Fortunately for us, Rails has found a nice balance in the form of its plugin architecture.

This book focuses on how you, as a developer, can leverage plugins to accelerate the development of your own projects. We'll demonstrate how major features can be added to existing applications with very little effort through a series of specific examples, allowing you to implement features that would traditionally take you far, far longer to develop on your own. We'll also show you how to create your own plugins to share with others. This chapter will get you started by giving you an idea of what plugins can do for you and then explaining how to find, install, and manage plugins.

What Can Plugins Do for You?

Plugins, in a nutshell, are bits of packaged code that provide either a modification to existing framework behavior or entirely new functionality. Some plugins are simple things that just make a new helper available in your views, such as the Gravatar plugin (`http://gravatarplugin.rubyforge.org`), which allows you to display a user's existing avatar icon from the popular Gravatar web service. On the other hand, some involve more complex functionality, such as adding geolocation capabilities to the default Active Record finder (GeoKit, Chapter 14) or providing a complete login/authentication system and user model (Restful Authentication, Chapter 11). The following are just some of things plugins can do:

- Extend an existing data model, allowing any resource to have items like ratings, tags, or comments associated with it, or allowing the model to act as something else entirely, such as a list or a tree data structure.
- Provide whole application subsystems, such as user authentication and authorization.

- Enhance your application views by wrapping easy-to-use helpers around popular third-party packages for graphing, media playback, web services access, and so on.
- Add Rake tasks to your Rails project, such as one to generate code coverage statistics or data model annotations.
- Change the way testing or other built-in facilities work.

As you can see, plugins serve a multitude of purposes. The plugin architecture in Rails is itself deceivingly simple, and the real magic comes from the metaprogramming capabilities inherent in an elegant dynamic language like Ruby, which allows any class to be opened and modified at runtime. This means that it's possible to change or extend any and all of the default behaviors in your models, controllers, and views.

Plugins are often used as a way to experiment with new ideas. In fact, features that eventually become part of the Rails core codebase can often begin life as plugins, as they provide a mechanism with which to share new ideas and functionality without directly modifying Rails itself.

Other features introduced as plugins simply have no place in the Rails core and are intentionally left out in the interest of keeping the core as general purpose and as slimmed-down as possible. These plugins serve purposes as diverse as payment processing (ActiveMerchant, Chapter 10), lightboxing images (Lightbox Helper, Chapter 19), testing HTML markup validity (Assert Valid Markup, Chapter 27), and interfacing with message queues (Workling, Chapter 18). They demonstrate exactly why the system works as well as it does; you can create a new project and elect to install only the specific pieces that make sense for your project, without worrying about the bloat of unused features. These less general-purpose plugins will remain plugins forever, existing as segmented architectural bits that can be maintained and updated on their own release schedule.

Like the basic foundation of open source software itself, the plugin architecture leverages the collective intelligence and experience of the entire Ruby on Rails community. Do you need to generate PDF documents on the fly? Bruce Williams and Wiebe Cazamier have written a tool, in the form of a plugin, that allows you to use LaTeX to do just this (RTex, Chapter 24). Do you need to handle image file uploads, create thumbnails, and store them at Amazon S3 (Simple Storage Service)? Rick Olson has already devised a way to do all that for you, too (Attachment Fu, Chapter 3).

The contributions made by the community in this area are staggering. Indeed, the popular plugin directory service at `http://agilewebdevelopment.com` counts more than 700 plugins in existence today, with new code being added by talented Rails developers every day. We can't cover them all here, but what we can do is get you started. However, before we can start exploring all the great uses for plugins and demonstrating examples with our sample applications, we first need to discuss the basic tools for discovering and installing plugins.

Managing a Wealth of Plugins

As you're no doubt aware, when you create a new Rails project, a number of helpful scripts are automatically available to you. One of these scripts is the plugin manager, available in `script/plugin` within the Rails project structure. This script is the gateway through which plugins can be discovered, installed, updated, and removed.

■ **Note** We assume that you already have Rails installed and you are working with version 2.0.2 or later. If you don't already have Rails installed, refer to the instructions at `http://rubyonrails.org` or the book *Beginning Rails: From Notice to Professional* by Jeffrey Allan Hardy (Apress, 2007).

Let's create a new Rails project so we can start exploring the plugin system. Enter the following command:

```
rails plugin_explorer
```

Finding Available Plugins

Plugins are typically available through a series of plugin repositories. When you create a Rails application, the plugin manager comes with a predefined set of defaults. Change to the newly created project directory and issue the following command to tell the plugin manager to list the default set of plugins available to a virgin Rails project.

```
cd plugin_explorer
ruby script/plugin list
```

```
account_location              ➥
http://dev.rubyonrails.com/svn/rails/plugins/account_location/
acts_as_list                  ➥
http://dev.rubyonrails.com/svn/rails/plugins/acts_as_list/
acts_as_nested_set            ➥
http://dev.rubyonrails.com/svn/rails/plugins/acts_as_nested_set/
acts_as_tree                  ➥
http://dev.rubyonrails.com/svn/rails/plugins/acts_as_tree/
atom_feed_helper              ➥
http://dev.rubyonrails.com/svn/rails/plugins/atom_feed_helper/
auto_complete                 ➥
http://dev.rubyonrails.com/svn/rails/plugins/auto_complete/
continuous_builder            ➥
http://dev.rubyonrails.com/svn/rails/plugins/continuous_builder/
deadlock_retry                ➥
http://dev.rubyonrails.com/svn/rails/plugins/deadlock_retry/
in_place_editing              ➥
http://dev.rubyonrails.com/svn/rails/plugins/in_place_editing/
javascript_test               ➥
http://dev.rubyonrails.com/svn/rails/plugins/javascript_test/
legacy                        ➥
http://dev.rubyonrails.com/svn/rails/plugins/legacy/
localization                  ➥
http://dev.rubyonrails.com/svn/rails/plugins/localization/
open_id_authentication        ➥
http://dev.rubyonrails.com/svn/rails/plugins/open_id_authentication/
```

```
scaffolding                    ➥
http://dev.rubyonrails.com/svn/rails/plugins/scaffolding/
scriptaculous_slider           ➥
http://dev.rubyonrails.com/svn/rails/plugins/scriptaculous_slider/
ssl_requirement                ➥
http://dev.rubyonrails.com/svn/rails/plugins/ssl_requirement/
token_generator                ➥
http://dev.rubyonrails.com/svn/rails/plugins/token_generator/
tzinfo_timezone                ➥
http://dev.rubyonrails.com/svn/rails/plugins/tzinfo_timezone/
tztime                         ➥
http://dev.rubyonrails.com/svn/rails/plugins/tztime/
upload_progress                ➥
http://dev.rubyonrails.com/svn/rails/plugins/upload_progress/
```

There are some interesting plugins in this list, but note that they're all in the same repository. Also note that the plugin repository itself is really just a Subversion (`svn`) source control repository, accessible via the HTTP protocol. You can open your web browser and plug in one of these URLs if you like. You'll see a directory listing that corresponds to the current Subversion revision for that plugin source. Feel free to read the plugin's `README` file or poke around in its source, but don't get sidetracked; we'll be talking about how plugins are internally structured in Chapter 28, where you'll learn how to create, test, and distribute your own plugin.

Tip The list of sources is shared across projects and is stored in a file called `.rails-plugin-sources` in your home directory. If you see additional plugins or repositories in this list, it's most likely because you previously added them using the `source` command.

What if you want to add another repository in order to expand the list of available plugins? You can use the plugin manager's `discover` and `source` commands. If you use the `discover` command, the plugin manager will ask you whether you want to add repositories from a known list to your personal set.

```
ruby script/plugin discover
```

```
Add http://www.agilewebdevelopment.com/plugins/? [Y/n]
Add svn://rubyforge.org/var/svn/expressica/plugins/? [Y/n]
Add http://soen.ca/svn/projects/rails/plugins/? [Y/n]
...
```

If you rerun the `list` command, you'll notice that the list of plugins has been greatly expanded. It now includes all of the plugins located within the new repositories that you've discovered.

Note The list of repositories retrieved by the `discover` command is actually scraped from the following URL: `http://wiki.rubyonrails.org/rails/pages/Plugins`. Since this page is a wiki and it's publicly editable by just about anyone, you may want to use discretion when adding plugins. In general, it's not a good idea to install plugins named things like `acts_as_identity_theft`! If you add a repository and want to later remove it, you can use the `unsource` command.

You can manually add plugin repositories to the local set by using the `source` command. Try this:

```
ruby script/plugin source http://svn.railsplugins.com/plugins
```

```
Added 1 repositories.
```

If you want to just list plugins available at that repository, use the `list` command with the `--source` option.

It's important to note that sourcing a repository is not a prerequisite for installing plugins from that repository. When installing plugins, you can specify either the full path to the plugin or a short name for the plugin if the plugin's repository is in your sources list. We'll cover how to install plugins next.

RUBY GEMS

In addition to plugins, we'll be using a number of useful Ruby gems throughout this book. RubyGems is a general package management system for Ruby libraries, and is not specific to Rails. Gems and plugins cover a lot of the same ground, with the primary differences being internal structure and the fact that plugins are specific to Rails, following certain conventions that allow them to hook into the Rails initialization process. Often, you'll find that a Rails plugin will consist of a thin wrapper around a more powerful gem.

Gems have many features that are lacking in Rails plugins, including dependency management and proper versioning, and there are ongoing discussions about bridging this divide. In the meantime, if your application depends on any number of gems, you may wish to unpack them to the `vendor` directory to ease deployment and versioning concerns.

To make this easier, Rails 2.1 has added a mechanism for defining and associating gem dependencies with Rails projects using the `config.gem` directive in `environment.rb` and the `rake gem` family of Rake tasks. Running `rake gems:install` will install gems that are listed as dependencies on the local system, and `rake gems:unpack` will unpack those gems to the `vendor` directory automatically. For more information, see the excellent Railscast screencast resource at http://railscasts.com/episodes/110. If you're using Rails 2.0 or older, we recommend checking out the Gems On Rails plugin by Dr. Nic Williams, which provides equivalent functionality. Gems On Rails is available at `http://gemsonrails.rubyforge.org`.

For more information about Ruby gems in general, check out *Practical Ruby Gems* by David Berube (Apress, 2007).

Installing Plugins

A plugin can be installed by using the plugin manager's `install` command. Once installed, integration with your application can often be as simple as adding a line to your `environment.rb` configuration or adding declarative-looking statements to an existing model class.

To demonstrate how to install Rails plugins, as well as see how they work in practice, let's install a couple extremely useful plugins. First, we'll examine the simple Annotate Models plugin, by Dave Thomas, and then we'll take a brief look at a slightly more complex plugin, Exception Notification, by Rails core team member Jamis Buck.

Installing the Annotate Models Plugin

The Annotate Models plugin adds a comment summarizing the current database schema to the top of each Active Record model source file. These annotations are very useful, since otherwise the table attributes may not show up at all within the body of your model. It's certainly a best practice to have these attribute definitions handy when looking at your model source. Imagine adding new developers to your development team and making them examine SQL table schemas in order to understand your data models. Better to make those models as self-explanatory as possible, right?

If the plugin is listed in your local sources, you can simply issue a command like `script/plugin install annotate_models`. However, Annotate Models is not in your default local sources, and we don't want to bother to source the whole repository. Instead, we can just specify the full path to the plugin, including the repository:

```
ruby script/plugin  install http://svn.pragprog.com/Public/plugins/annotate_models
```

```
+ ./annotate_models/ChangeLog
+ ./annotate_models/README
+ ./annotate_models/lib/annotate_models.rb
+ ./annotate_models/tasks/annotate_models_tasks.rake
```

This output shows that the most recent version of the specified plugin has been checked out from its Subversion repository. It has been placed in the `vendor/plugins` directory of the project, which is where all per-project plugins are stored. Any plugins that are present within that directory structure are automatically loaded at startup. In this case, the Annotate Models plugin simply adds a Rake task to the default set of tasks for your Rails project. You can learn more about it by looking at the plugin's `README` file, which should now be located in your local source tree at `vendors/plugins/annotate_models/README`.

To test the newly installed plugin, you'll first need a model to annotate. Let's create one now. We'll generate a `Book` model that has some number of attributes.

```
ruby script/generate model book
```

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/book.rb
      create  test/unit/book_test.rb
      create  test/fixtures/books.yml
      create  db/migrate
      create  db/migrate/001_create_books.rb
```

Next, we'll edit the migration to specify a number of attributes. Edit the file `db/migrate/001_create_books.rb` so it looks like this:

```
class CreateBooks < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.column :title, :string
      t.column :isbn, :string
      t.column :author, :string
      t.column :publisher, :string
      t.column :description, :text
    end
  end

  def self.down
    drop_table :books
  end
end
```

As of Rails 2.0.2, any new Rails project uses SQLite version 3 as its default database back end unless otherwise specified. This means that when you run your database migration, a database is automatically created for you in `db/development.sqlite3`. If you would prefer to use MySQL or PostgreSQL, you can edit the `config/database.yml` file at this time.

Run the migration now by entering the following command:

```
rake db:migrate
```

```
== CreateBooks: migrating ====================================================
-- create_table(:books)
   -> 0.8761s
== CreateBooks: migrated (0.8762s) ===========================================
```

At this point, you have an application with a single data model, `Book`, which is, through the magic of Active Record, associated with the `books` table in your database. A `Book` instance

has five attributes: a title, an ISBN, an author, a publisher, and a description. If you open your model code in `app/models/book.rb`, you can see the model definition:

```
class Book < ActiveRecord::Base
end
```

That isn't very descriptive, is it? Let's try out the Annotate Models plugin, to see how that can help us here. If you viewed the `README` file earlier (good initiative!), you already saw the instructions for use. It's very simple. The plugin provides a Rake task to annotate any model objects in the current project. Let's run it:

```
rake annotate_models
```

```
Annotating Book
```

That's all the output you'll see—nothing fancy. But if you open the model definition, you can see the results of this action:

```
# == Schema Information
# Schema version: 1
#
# Table name: books
#
#  id          :integer(11)   not null, primary key
#  title       :string(255)
#  isbn        :string(255)
#  author      :string(255)
#  publisher   :string(255)
#  description :text
#

class Book < ActiveRecord::Base
end
```

Now that's much clearer. Simply by looking at the model source, you can see which attributes are available. In addition, this information will be nice to have around if you plan on generating API documentation for this project with RDoc (the standard embedded documentation generator for Ruby).

As you can see, the Annotate Models plugin isn't at all complicated in terms of what it does. It is, however, universally useful and a great tool to have in your box of development tricks. As we examine numerous plugins throughout this book, you'll find that almost all of them do far more complicated things than Annotate Models, but few of them are as universally applicable across all projects.

Installing the Exception Notification Plugin

Moving on to a slightly more complicated example (which also has universal appeal), we'll next examine the Exception Notification plugin. If you have an application deployed in a production environment, exception notifications can be crucial to have in place.

Imagine that you're an experienced programmer learning Rails and you've just deployed your first big application. Unfortunately, your boss chewed you out this morning because for the past week, a big client was trying to use an obscure feature of the site that hadn't been properly tested by one of your subordinates. It turns out that a helper function was referencing a misnamed controller, which was causing an exception to be raised. Worse, the client waited an entire week to notify support about it. If you had had some warning, you could have fixed it before the complaint had escalated!

With exception notification in place, if your application experiences an unhandled exception, the details of that exception will be immediately packaged and sent to some e-mail address that you specify. The e-mail message will contain a wealth of information about the exception, including the current request, session, and environment data, in order to help you re-create and resolve the problem. It's an essential tool, particularly in the early production life stages of a new application.

First, let's install the plugin:

```
ruby script/plugin install ➥
http://dev.rubyonrails.org/svn/rails/plugins/exception_notification
```

```
+ ./exception_notification/README
+ ./exception_notification/init.rb
+ ./exception_notification/lib/exception_notifiable.rb
+ ./exception_notification/lib/exception_notifier.rb
+ ./exception_notification/lib/exception_notifier_helper.rb
+ ./exception_notification/test/exception_notifier_helper_test.rb
+ ./exception_notification/test/test_helper.rb
+ ./exception_notification/views/exception_notifier/_backtrace.rhtml
+ ./exception_notification/views/exception_notifier/_environment.rhtml
+ ./exception_notification/views/exception_notifier/_inspect_model.rhtml
+ ./exception_notification/views/exception_notifier/_request.rhtml
+ ./exception_notification/views/exception_notifier/_session.rhtml
+ ./exception_notification/views/exception_notifier/_title.rhtml
+ ./exception_notification/views/exception_notifier/exception_notification.rhtml
```

As before, the plugin has been checked out to your `vendor/plugins` directory, where it will be loaded on startup. It's important to note that if you install new plugins while you're running a development server, you'll need to restart the server. Plugins are loaded only on startup, even in development mode. For the Annotate Models plugin, this didn't matter so much, because the plugin was providing only a Rake task. In this case, the Exception Notification plugin will be mixed into your `ApplicationController`, extending Rails itself, so the server will need to be restarted before things will work properly.

Exception Notification, like many plugins, requires some minimal amount of integration into your project to function. This is typical of plugin installations. Although the plugin is already installed, you need to tell it how to talk to the rest of the application. In most cases, this means adding a line or two of declarative-looking code to a model class, or setting a configuration directive in `environment.rb`. The specifics depend on the plugin and its purpose. The `README` file that comes with the plugin will always document this process, so be sure to

read it. In the case of Exception Notification, you'll need to include the `ExceptionNotifiable` mixin in your `ApplicationController` (`app/controllers/application.rb`):

```
class ApplicationController < ActionController::Base
  include ExceptionNotifiable
  ...
end
```

The act of mixing in this new module will make sure that the application fires off a notification e-mail whenever an exception is encountered. You also need to specify the e-mail recipients in your `config/environment.rb` file:

```
ExceptionNotifier.exception_recipients = %w(support@your-company.com)
```

You can further customize the plugin if you wish, to set the sender address of the e-mail, the text used to prefix the subject line of the e-mail, and so on. See that `README` file for details.

In any case, you can now restart the application. If it's running in production mode and an exception is generated from a request source other than localhost (that is, from any remote user), an e-mail will be sent to the intended recipient.

To test exception notifications while running locally in development mode, you need to tweak a few extra items. This is because Rails treats exceptions differently in production versus development mode (it also treats local requests specially). In production mode, we want to protect users from seeing the details of the exception that occurred. In development mode, we'll display any and all exception data to the user in an error template. These differences are reflected in the `rescue_action_in_public` and `rescue_action_locally` methods of the `ActionController::Rescue` module, and Exception Notification, being interested only in remote requests received while in production mode, hooks into only the former method.

To remedy this, so we can test the plugin locally, add the following two lines to the end of your `ApplicationController` class definition in `app/controllers/application.rb` (immediately before the `end`).

```
local_addresses.clear
alias :rescue_action_locally :rescue_action_in_public
```

■ **Note** What this does is essentially force the requests that result in exceptions in the development environment to be handled the way that they would be handled for remote client requests in production, which, since we've mixed in `ExceptionNotifiable`, involves sending the exception notification e-mails. We've also told Rails to treat all requests as nonlocal—even requests from localhost. Again, this is for the purpose of testing our plugin only, and you'll probably want to remove this code after verifying that exception notifications are working. In most cases, you would prefer to see the standard `rescue_action_locally` behavior while in development, since it will display the error in your browser window and allow for quicker debugging.

Now you need to add something to your application to generate an exception. That should be easy enough, right? We've all done it. Edit your `config/routes.rb` file and add the following line before the default route specification:

```
map.connect '/foo', :controller => 'foo'
```

Since there is no `FooController`, this should cause an exception (in this case, a `NameError`) to be raised. If everything is configured correctly, your application will write the notification to your development log. To see this for yourself, start your server using `ruby script/server` and open a web browser. Surf to `http://localhost:3000/foo`, and you should be greeted with the default 500 Server Error page. If you look in your `log/development.log`, you'll see that the exception notification has been generated. It should look something like this:

```
Sent mail:
 From: Exception Notifier <exception.notifier@default.com>
Subject: [ERROR] application#index (NameError) ➥
"uninitialized constant FooController"
Mime-Version: 1.0
Content-Type: text/plain; charset=utf-8

A NameError occurred in application#index:

  uninitialized constant FooController
  /opt/local/lib/ruby/gems/1.8/gems/activesupport-
1.4.2/lib/active_support/dependencies.rb:266:in ➥
'load_missing_constant'
...
```

In production mode, assuming that Action Mailer is configured correctly, an e-mail alert will be sent. The e-mail message will contain this information along with the subsequent backtrace, session, and environment information.

Updating and Removing Plugins

It's important to note that once a plugin is installed, it's essentially locked to the version you installed (unless you use `svn:externals`, which will be discussed shortly). This means that if there are later updates to the plugin source, you'll need to explicitly update the plugin to receive those updates. You can do this using the plugin manager's `update` command. To update the Exception Notification plugin, for example, you would issue the following command:

```
ruby script/plugin update exception_notification
```

Similarly, if you ever wish to uninstall the Exception Notification plugin, or any plugin at all, you can use the `script/plugin remove` command. For more information and extended options, you can always issue the following command to receive the default help message:

```
ruby script/plugin -h
```

RaPT

RaPT, the Rails Plugin Tool, is a more feature-rich replacement for the default Rails plugin manager. Perhaps its most notable addition is the ability to search plugins from the command line. RaPT is available as a gem, and you can install it by issuing the following command:

```
gem install rapt
```

Let's run a search with RaPT. It's very similar to the way that Debian's `apt-get` tools and various other package management tools work:

```
rapt search "gravatar"
Gravatar
  Info: http://agilewebdevelopment.com/plugins/show/689
  Install: svn://rubyforge.org//var/svn/gravatarplugin/plugins/gravatar
Gravatar tag
  Info: http://agilewebdevelopment.com/plugins/show/783
  Install: http://tools.assembla.com/svn/hasham/plugins/gravatar_tag
```

RaPT also has support for *plugin packs*, a term used to describe sets of plugins that are grouped together for ease of installation. You can learn more about plugin packs (including how to build one of your own) at `http://www.lukeredpath.co.uk/2006/7/27/install-your-favorite-rails-plugins-easily-with-rails-plugin-packs`.

The other RaPT commands are analogous to the standard plugin managers, using the same syntax for installing plugins, updating them, and so on. In short, RaPT does everything the stock plugin manager does and then some. More information is available at `http://rapt.rubyforge.org`.

Accessing Documentation and Tests

Documentation and testing are just as important for plugins as they are for any other piece of code. Fortunately, Rails provides a couple of default Rake tasks that deal with running plugin tests and generating documentation for the plugins you have installed. You'll learn about writing tests for our own plugins in Chapter 28.

To find out what Rake-related plugin tasks are available, you can issue the following command:

```
rake -T | grep plugin
```

```
rake doc:clobber_plugins       # Remove plugin documentation
rake doc:plugins               # Generate documentation for all installed plugins
rake test:plugins              # Run the plugin tests in vendor/plugins/**/test ➥
(or specify with PLUGIN=name)
```

Let's generate some documentation for your installed plugins. It's simple:

```
rake doc:plugins
```

You'll see the familiar RDoc output generated. When it completes, the report shows the HTML output statistics, including the number of files, classes, modules, and methods processed for each plugin. The task deposits documentation for all your plugins in the `doc/plugins` directory. If you navigate there in a browser, you can open the HTML files and browse through the autogenerated API documentation for the plugin, as shown in Figure 1-1.

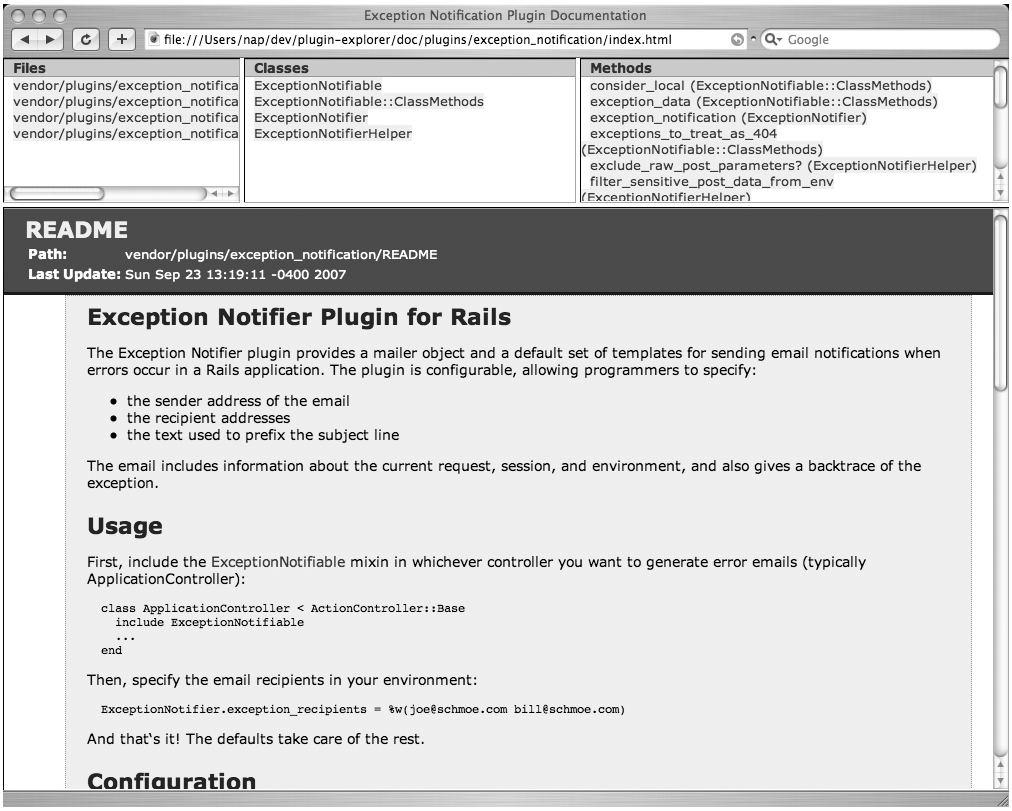

Figure 1-1. *Generated documentation for the Exception Notification plugin*

Running tests for installed plugins should also be familiar to anyone who is used to writing and running unit tests in Rails. To run all the tests that come packaged with your installed plugins, issue the following command:

```
rake test:plugins
```

```
Started
.......
Finished in 0.228314 seconds.

7 tests, 8 assertions, 0 failures, 0 errors
```

If you see this output, it means that everything is installed and working correctly.

Versioning Plugins

We've mentioned that most plugin repositories are really just Subversion repositories, and when you're installing a plugin, you're really just using Subversion to check it out of a remote Subversion repository and into your local source tree. And to be honest, that's really all you need to know about Subversion to get started. However, if you're already using Subversion for version control of your project, you can benefit greatly by knowing a little more about how it can help you with plugin management.

■ **Tip** As of Rails 2.1, plugins can also be installed directly from Git repositories. Git is a distributed version control system and is now used for development of Rails itself. For more information about Git, see http://git.or.cz.

Using svn:externals

Rails developers often use `svn:externals` to manage plugins in a Rails project. By setting the `svn:externals` property on your `vendor` directory, you're essentially telling Subversion that when your project is checked out, the code in `vendor` is an external dependency and should be fetched from the appropriate external third-party repository.

There are a couple benefits to doing this. First, it means that you don't need to store the plugin code in your own repository—the code will always be pulled from the actual plugin source. This means that whenever the plugin is updated, you'll reap the benefits of those new features immediately (the next time you `svn update` your source).

You can use the handy `-x` argument to the `plugin install` command to tell the plugin manager to install the plugin and add an `svn:externals` entry to `vendor/plugins`:

```
ruby script/plugin install -x exception_notification
```

Despite the obvious benefits to this approach, there are a few drawbacks you should be cognizant of as well. We mentioned before that whenever your project is checked out of your Subversion repository or updated, the latest plugin updates will also be pulled down automatically. This is great in a lot of cases, but it can also be problematic if the updated plugin unexpectedly breaks something in your codebase. To overcome this, you can manually add the `-r` argument to specify that you want the revision number locked.

Another drawback is what happens if the third-party repository becomes temporarily unavailable, which is a liability many businesses just can't afford. And of course, if you're working in a team environment, as most of us are, and dealing with a high volume of Subversion updates to keep in sync, you must remember that every time an update is issued, Subversion is checking not only your local repository but also each remote plugin repository for updates.

Oh, and what happens if you decide you need to make a change to some of that plugin code? Smells like trouble.

■ **Note** If you insist on using `svn:externals` anyway, note that you can use the `-ignore-externals` option to the `update` command to ignore external repository checking.

Using Piston

Another solution, which has become increasingly popular and is arguably the best of both worlds, is to use the Piston gem to manage your plugins with Subversion. Piston is a Ruby gem that simplifies vendor branch management for plugins. To install it, type this:

```
gem install piston -y
```

Note that your project must be in a working copy of a Subversion repository in order for this to work.

Here's how you would use Piston to install the Annotate Models plugin, rather than using the stock plugin manager for the installation. Note that you can also specify a specific revision to snatch:

```
piston import http://svn.pragprog.com/Public/plugins/annotate_models ➥
vendor/plugins/annotate_models -r 15
```

```
Exported r15 from 'http://svn.pragprog.com/Public/plugins/annotate_models' to ➥
'vendor/plugins/annotate_models'
```

So what does Piston do that makes it so different? Piston essentially copies the current (or specified) revision of the imported plugin into your repository, and keeps track of the revision number and where it came from. You end up with a local copy of the plugin code checked into your repository, which you can modify if necessary. Later, you can choose to update those plugins by issuing a `piston update` command, which will replace that plugin source with a newer version from the vendor, and even manage merging your own modifications into it if necessary. Slick, eh?

```
piston update vendor/plugins/annotate_models/
```

```
Processing 'vendor/plugins/annotate_models/'...
  Fetching remote repository's latest revision and UUID
  Restoring remote repository to known state at r15
  Updating remote repository to r22
  Processing adds/deletes
  Removing temporary files / folders
  Updating Piston properties
  Updated to r22 (3 changes)
```

Piston is extremely easy to use and is the plugin management strategy of choice for a growing number of professional Ruby on Rails developers.

If using Piston sounds good to you, but you have a large number of plugins already installed using `svn:externals`, you'll be pleased to know that Piston can also be used to convert existing plugins with its `convert` command. You can read more about Piston and its features at `http://piston.rubyforge.org`.

Throughout this book, we'll be using the standard plugin manager to install plugins because of its ubiquity and lack of dependence on Subversion. That said, we use Piston in real-world projects ourselves, and we think that you probably should, too.

Discovering Plugins

A commonly asked question is, "How do I find useful new plugins?" Running the `discover` and `list` commands from the plugin manager finds the names of available plugins through known repositories, and RaPT is a great tool to use when searching plugins for a particular keyword. However, there are other resources on the Web that are rich with useful information, allowing you to view lists of plugins by categories and read descriptions and user comments.

The first resource worth mentioning is, of course, the official Ruby on Rails Wiki, which maintains a page on plugins, separating them by category. You can view this page at `http://wiki.rubyonrails.com/rails/pages/Plugins`.

Another useful resource is the Agile Web Development Rails Plugin Directory, created by Ben Curtis, an exhaustive and well-organized directory of plugins. It is divided by categories and fully searchable. Users of the service contribute ratings and comments to each plugin entry, making it easy to filter out the great from the very good. RSS feeds are also available to keep you abreast of current developments. The Plugin Directory can be accessed at `http://agilewebdevelopment.com/plugins`. RailsLodge (`http://railslodge.com`) and RailsPlugins.net (`http://railsplugins.net`) provide similar offerings.

Finally, there is the web site associated with this book, `http://www.railsplugins.com`, which contains source code for the examples found here and supplementary information that you may find useful. You can also find the source code for the examples for this book in the Source Code/Download area of the Apress web site (`http://www.apress.com`).

Summary

Rails plugins put a wealth of powerful, multipurpose extensions at your fingertips. With a uniform installation mechanism, you only need to know how to look for them and employ a simple set of commands to get up and running quickly and easily, adding powerful functionality to your applications. Learning Subversion (or, alternatively, the even more powerful Git distributed version control system) is paramount, and most Rails developers are already using it for source control. If such is the case, learning to use Piston is very little extra work for a lot of extra flexibility in managing your plugins.

This book will demonstrate how you can quickly bring a web application into existence with minimal time spent on the more generic aspects such as login systems, dealing with attachments, user ratings, and such. By leveraging plugins, you can free yourself from these routine tasks and focus on those unique aspects that make your application stand out from the competition.